AGARD-AG-311

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARDograph No.311

COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS

by

W.Gentzsch
Rongtgenstrasse 42
D-8402 Neutraubling
Federal Republic of Germany

and

K.W.Neves
Manager, R & D Programs
Boeing Computer Services
12824 NE 135th St.
Kirkland, WA 98034
United States

Edited by

H.Yoshihara
Boeing Military Airplane Company
A Division of the Boeing Company
Mail Stop 33—18
P.O. Box 3707-2207
Seattle — WA 98124

This AGARDograph has been produced at the request of the Fluid Dynamics Panel of AGARD.

THE MISSION OF AGARD

According to its Charter, the mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

— Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community;

— Providing scientific and technical advice and assistance to the Military Committee in the field of aerospace research and development (with particular regard to its military application);

— Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;

— Improving the co-operation among member nations in aerospace research and development;

— Exchange of scientific and technical information;

— Providing assistance to member nations for the purpose of increasing their scientific and technical potential;

— Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

The content of this publication has been reproduced
directly from material supplied by AGARD or the authors.

Published March 1988

Copyright © AGARD 1988
All Rights Reserved

ISBN 92-835-0448-8

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

FOREWORD

Recent results have demonstrated the potential viability of current Navier Stokes computer programs to simulate complex viscous/vortical flows over aerospace vehicles over the flight spectrum. Such impressive demonstrations could not have been achieved without the advanced numerical algorithms and the powerful supercomputers such as the CRAY CYBER series. Although improved modelling of the flow equations is still needed, the major obstacle to evolving a cost-effective design tool is the large computing costs associated with the [illegible] Navier/Stokes equations. Despite the more powerful computers with multiple CPUs on the horizon, there is a need to develop still more cost-effective algorithms and to programme them in a manner that fully utilises the powerful vector hardware features of the supercomputers on hand. To accomplish the latter, an intimate knowledge of the supercomputer hardware and its impact on the algorithm is essential, and this will be the subject of the present AGARDograph.

Though the present AGARDograph is primarily directed to the active computational fluid dynamicist, it will serve as a valuable guide for computational researchers in other disciplines as well as for engineering managers. Of particular value is the glossary of supercomputer terms given in the Appendix.

The co-authors are Professor Dr Wolfgang Gentzsch, Technical University at Regensburg, and Dr Ken Neves, Boeing Computer Services in Seattle. Both authors have extensive experience with supercomputing, having detailed knowledge of the hardware features of current supercomputers and their impact on algorithms. (Their biographies follow on the next page.)

The AGARD Fluid Dynamics Panel and the Editor wish to express their appreciation to Professor Dr Gentzsch and Dr Neves for their dedicated effort in producing this timely AGARDograph. Dr J.Steger, NASA-Ames Research Center kindly contributed a Section in Chapter 6.

H.Yoshihara
Editor

* * *

AVANT-PROPOS

De récents résultats ont démontré la viabilité potentielle des programmes d'ordinateur de Navier Stokes actuels destinés à simuler des écoulements visqueux et tourbillonnaires complexes sur les véhicules aérospatiaux dans tout le spectre de vol. Ces impressionnantes démonstrations n'auraient pas pu être réalisées sans les algorithms numériques avancés et les superordinateurs puissants comme ceux de la série CRAY CYBER. Bien qu'une modélisation améliorée des équations d'écoulement soit encore nécessaire, le principle obstacle a l'évolution d'un outil d'etude rentable est constitué par les coûts de calcul élevés liés aux équations difficiles de Navier Stokes. En dépit du fait que des ordinateurs plus puissants avec de multiples unités centrales apparaissent à l'horizon, il est nécessaire de développer des algorithmes encore plus rentables et de les programmer de manière à utiliser entièrement les caractéristiques du puissant matériel des superordinateurs disponibles. Pour y parvenir, une connaissance intime du matériel des superordinateurs et de son impact sur l'algorithme est indispensable et tel sera le sujet du présent AGARDograph.

Bien que le présent AGARDograph soit principalement destiné au spécialiste actif du calcul de la dynamique des fluides, il constituera un guide précieux à l'usage des chercheurs pour le calcul dans d'autres disciplines ainsi qu'aux directeurs techniques. Le glossaire des termes relatifs aux superordinateurs figurant en annexe est d'un intérêt particulier.

Les co-auteurs sont le Professeur Dr Wolfgang Gentzsch, de l'Université Technique de Regensburg, et le Dr Ken Neves, des Services d'Informatique de Boeing, à Seattle. Les deux auteurs possèdent une vaste expérience en matière de calculs très poussés, du fait qu'ils ont une connaissance détaillée des caractéristiques du matériel des superordinateurs actuels et de leur impact sur les algorithmes. (Leurs biographies figurent à la page suivante.)

Le Groupe de Spécialistes de la Dynamique des Fluides de l'AGARD et le Rédacteur en Chef tiennent à exprimer leurs remerciements au Professeur Dr Gentzsch et au Dr Neves pour les efforts qu'ils ont déployés afin de réaliser en temps voulu cet AGARDograph. Le Dr J.Steger, du Centre de Recherche de la NASA-Ames, a bien voulu rédiger un Paragraphe du Chapitre 6.

H.Yoshihara
Rédacteur en Chef

BIOGRAPHY

Prof. Dr Wolfgang Gentzsch

Professor Gentzsch is presently Professor of Applied Mathematics at the Fachhochschule-Regensburg and is a consultant on evaluating the suitability and potential of various supercomputer architectures for various applications. Prior to 1985 he worked at several research institutes, most notably the Max Planck Institute for Plasma Physics (Garching) and the German Research Establishment for Aeronautics and Astronautics (DFVLR-Göttingen). At DFVLR Professor Gentzsch was Head of the Computational Fluid Dynamics Group (1981–1985) specializing in supercomputer software development and restructuring CFD algorithms for vector computers.

Professor Gentzsch received the Diploma in Mathematics/Physics from the Technical University-Aachen in 1972 and the Doctors Degree in Numerical Mathematics from the Technical University-Darmstadt.

Dr Kenneth W.Neves

Dr Neves is currently Manager of Research and Development Programs in the Boeing Computer Services Company (BCS) in Seattle, Washington. His primary responsibilities include the definition and management of research and development programs encompassing most aspects of engineering/scientific computing, notably hardware, software, and algorithms. A major activity under Dr Neves' direction is the High Speed Computing Program which currently has 3 parallel processors and advanced workstation equipment.

Previously he was Manager of the Computational Mathematics Group responsible for maintenance, development, and certification of mathematical and statistical software libraries resident on the BCS computer centers nationwide. Before joining BCS in 1975, Dr Neves was Senior Mathematician for the Nuclear Power Division of Babcock and Wilcox Company, Lynchburg, Virginia.

He is currently Vice-Chair and co-founder of the SIAM Special Interest Group on Supercomputing, the Chair and founder of the Special Interest Committee on Applications and Algorithms of the Cray User Group, and serves on the IEEE Subcommittee on Supercomputing.

Dr Neves received the B.A. Degree in Mathematics (with great distinction) from California State University at San Jose, California, and received the M.A. and Ph.D. Degrees in Mathematics (specializing in numerical analysis) from Arizona State University while holding a National Science Foundation Research Fellowship.

CONTENTS

Chapter 1. BACKGROUND AND OVERVIEW (Editor)

1.1 INTRODUCTION

The arrival of the supercomputer together with the development of efficient algorithms have made feasible computations of important fluid dynamic problems arising in high performance aircraft. Two current problems that contain many of the significant fluid dynamic mechanisms are as follows. The first concerns the transonic maneuverability of advanced combat aircraft required by survival and weapons delivery considerations. Such aircraft utilize leading edge vortices and vectored thrust to generate high lifts with tolerable drags. (See Fig. 1) Maneuver performance is limited by the bursting of the vortices and severe shock-induced separation on the wing that lead to vehicle dynamic instabilities. In relevent cases, the boundary layer is essentially turbulent. The second problem concerns the high altitude cruise and reentry performance of hypersonic aircraft flying for example at a Mach number of 25 (Fig. 2). Here important features are strong shock waves that create large drags and large boundary layer energy dissipation that leads to strong surface heating. The air in the shock layer can further become sufficiently heated that the internal modes of the air molecules become excited, and reactions as the dissociation of the air molecules become significant. If a significant number of molecules has been dissociated, the catalytic wall effect can greatly increase the surface heating by enhancing the surface recombination and releasing the heat of recombination. Clearly, these effects greatly modify the flow field as a whole. At high hypersonic Mach numbers at high altitude conditions, the boundary layer over a significant portion of the configuration is laminar, and transitional boundary layers will play an important role.

For the proper treatment of these problems, the Reynolds-averaged Navier/Stokes (N/S) equations, made fully determinate with a suitable turbulence model, must be used. In the unsteady (perfect gas) formulation, the resulting equations with an algebraic turbulence model are of mixed parabolic/hyperbolic type, with the characteristics such that a marching solution procedure can be employed. If a steady solution is desired, it would be obtained as the limiting flow for large times. In the steady formulation, the N/S equations are also of mixed parabolic/hyperbolic type, but the characteristic slopes are such as to preclude a marching when both subsonic and supersonic flow regions are present. The above transonic problem must therefore be posed in the unsteady form if a marching solution procedure is to be used. In the hypersonic case, a marching of the steady equations would be prevented by the presence of the thin subsonic sublayer in the boundary layer. If a sublayer approximation is made where the pressure across the subsonic sublayer is assumed to be constant, the upwind-biased characteristic is removed; and a spatial marching in the downwind streamwise direction is then permitted. The steady equations with the sublayer approximations are called the parabolized N/S (PN/S) equations, and they have yielded solutions that have matched experiments in significant cases up to Mach numbers of 18.

The essential elements in numerically solving the above problems are the generation of a suitable mesh and a stable and accurate difference algorithm. The importance of the mesh generation cannot be over-emphasized, affecting both the accuracy and stability of the numerical procedure. Above all, the mesh must be adequately refined and orthogonal, particularly in the neighborhood of the configuration surface. Additionally, to ease the fulfillment of the boundary conditions, the mesh should be conformal to the configuration; and this is usually accomplished by a curvilinear transformation which maps the boundaries in the physical domain to the surface of a cube. The required mapping and the proper refinement and orthogonality of the mesh are determined, either by an algebraic interpolation scheme, or by solving an appropriately posed elliptic, hyperbolic, or parabolic differential equation problem. For complex configurations, as an advanced fighter, the flow domain is divided into a number of subdomains, each embedding a key component of the configuration. With this multi-block structure, the mesh generation in each block is greatly simplified, and the mesh can be efficiently tailored to the specific component. Use of the multi-block mesh will require additional programming to insure the proper continuity of the flow across the block boundaries.

1.2. NAVIER/STOKES METHODS

There are two classes of algorithms currently in use in solving the unsteady N/S problem. These are the explicit methods, using for example a Kutta-Runge difference scheme (Swanson and Turkel [1.1]), and the implicit methods as the approximate factorization method (Steger-Pulliam [1.2]). Explicit methods are simple and suited for the vector (pipeline) computers, but are hampered by stringent linear stability limits on the marching step. Multigrid methods have been investigated to reduce the large computing times with the explicit methods, but the extreme aspect ratio of the boundary layer mesh have precluded the success experienced with the Euler equations. Perhaps a multi-block mesh should be employed isolating the boundary layer and confining the use of the multigrid to the "inviscid" mesh blocks in which the large wave length (hard-to-damp) components of the truncation error occur.

In the implicit case, with centered second order spacial differencing, the problem reduces to the inversion of a block septa-diagonal matrix, the six off-diagonal blocks arising by the centered differences in the three spacial directions. The dimension of each block is 5 x 5 corresponding to the five conservation equations. In the case of the Steger-Pulliam ARC3D code [1.2], the Beam/Warming approximate factorization is employed to reduce the block septa-diagonal matrix to a product of three block tri-diagonal matrices, which can then be more readily inverted. The Beam/Warming implicit difference scheme in three space dimensions is unstable, so that in the ARC3D code an implicit second order and an explicit fourth order artificial damping were employed. With the proper choice of the damping coefficients, the resulting Beam/Warming scheme becomes linearly unconditionally stable. However, a limitation on the marching step arises due to nonlinear instabilities, but the limitation here is in most cases significantly less severe than the CFL conditions in the explicit schemes.

Subsequently, further improvements of the ARC3D code were made that greatly reduced the computing time (Pulliam and Steger [1.3]). One significant improvement was the (approximate) reduction of the block tri-diagonal Jacobian matrices to scalar tri-diagonal matrices using the matrix eigenvectors. This reduction to scalar matrices then permitted the use of the more effective fourth order implicit damping instead of the second order damping, since the resulting scalar penta-diagonal matrices could be inverted with modest additional computing.

Mention must be made of the MacCormack two-step (predictor/corrector) method [1.4], an earlier method still widely used. It utilizes the Lax/Wendroff two-step explicit difference scheme in regions where the CFL stability restrictions are not severe as in inviscid interior flow, but switches the second (corrector) step to an implicit scheme where the marching step restrictions are severe as within the boundary layer. The implicit step here requires the inversion of a block bidiagonal matrix. Use of the MacCormack method has been restricted by the inflexibility to impose boundary conditions of interest in the implicit step.

More recently, methods have been in development in an attempt to reduce the computing time. Here the block septa-diagonal matrix was inverted directly using a line Gauss/Seidel (relaxation) procedure (Thomas and Walters [1.5] and MacCormack [1.6]). Here the performance of the line Gauss/Seidel procedure will depend on the rate of convergence of the relaxation process which in turn will depend on the dominance of the diagonal blocks. Experience to date for large problems has not shown improvements of the computing times with the Gauss/Seidel relaxation method.

There have been refinements in the difference schemes (upwind schemes) where the numerical viscosity was automatically tailored to improve the shock capture. These include the use of flux limiters in the eigenvalue split-flux methods and the total variation diminishing (TVD) methods. These techniques have been successful in removing shock "wiggles", particularly for strong shocks, but have been unable to reduce the capture thickness of highly swept shocks. More importantly, a proper upwind differencing precludes the need of additional artificial viscosity, though of course the greater flexibility with the latter is lost assuming the proper assignment of the "coefficient of viscosity" can be made. Recent unsteady N/S methods of Thomas and Walters [1.5] and Ying, Steger, Schiff, and Baganoff [1.7] have employed such upwind difference schemes. Computing time was however increased significantly.

1.3. EXAMPLES

The present status of N/S calculations for the transonic and hypersonic problems is next illustrated by several examples.

1.3.1 Transonic Case

In the transonic case, the first examples include the turbulent flow calculations of two swept wings by Kaynak, Holst, and Cantwell [1.8] using the transonic N/S (TN/S) code; that is, the improved ARC3D code with a multi-block mesh. The first case is for the 20^o-sweep wing with a NACA 0012 airfoil section for a Mach number of 0.826, an angle of attack of 2^o, and a Reynolds number based on the wing chord of 8×10^6. In Fig. 3 the calculated chordwise pressure distributions at three span stations are compared with experimental results. Two turbulence models were used: the Baldwin/Lomax mixing length model and its lag extension. The inadequate shock/boundary layer interaction is evident here and has resulted in an insufficient weakening of the terminating shock. This then leads to an erroneous shock location and to errors in the forces and moments on the wing. The airfoil results of Johnson (Ref. 9) suggest the cause to be an inadequacy of the algebraic turbulence model which lacks the proper lagging of the turbulent coefficient through the shock pressure-rise. With the proper lag, a larger displacement wedge at the base of the shock would result, producing the greater weakening of the shock to match the experiments.

The second case (also from [1.8]) is for a 45^o-sweep, tapered and twisted wing at a Mach number of 0.9, an angle of attack of 5^o, and a Reynolds number based on the mean aerodynamic chord of 6.8×10^6 In Fig. 4 the chordwise pressure distributions are compared to the experimental results, and good agreement is achieved except in the two outboard stations. At the 70% semi-span station, both the forward and the rear shocks are inadequately captured, whereas at the 90% station, the outboard shock is badly captured. The inability to capture the swept forward shock is a frequent failing due to the failure to refine the mesh, not only in the streamwise direction, but in the transverse directions also cutting across the shock. The improved capture of the stronger rear or outboard shock must await an improved modeling of the turbulent transport. In Fig. 5 we show the skin friction lines that were predicted, comparing them with a schematic of an oil-flow picture from the wind tunnel tests. Calculations have very impressively predicted the envelope of the skin friction lines forming the separation line. The location of the separation line was not predicted correctly due to the inadequate shock/boundary layer interaction described above. An open separation was predicted without the spiral vortices seen in the oil-flow picture, suggesting the need for an improved viscous modeling in the lower part of the boundary layer where a large cross-flow occurs. The consequences on the pressure and skin friction distributions due to the absence of the spiral vortices however may not be significant.

The second example illustrates recent accomplishments in the mesh generation for a complex configuration, in this case the General Dynamics F-16. In Fig. 6 is shown the multi-block mesh from [1.10] generated however in connection with an Euler calculation. Shown here in particular are the surface mesh and the block structure for the under-fuselage inlet. For a Navier/Stokes calculations, the mesh must be additionally refined near the configuration surface to resolve the boundary layer flow. The generation of a good mesh constitutes a major part of the problem solution, and the above results are significant accomplishments.

1.3.2 Hypersonic Case

Hypersonic calculations are less advanced than for the transonic case. In a recent example [1.11] the generic wing/fuselage shown in Fig. 7 was calculated using the NASA-Ames/Boeing parabolized N/S (PN/S) code which employed the Beam/Warming approximate factorization and the Baldwin/Lomax turbulence model. The Mach number was 25, the angle of attack 0°, the altitude 217,000 feet, and the surface temperature 1758° R. The air composition was assumed to be in equilibrium; that is, the reaction rates were assumed to be sufficiently rapid that the specie number densities assumed their equilibrium values corresponding to the local temperature and density. In Fig. 8 we show typical spanwise distributions of the pressure and heat transfer at various streamwise stations. At a given axial station, the peaking of the heat transfer is seen to be tied to the peaking of the pressure which occurs at the reattachment point. The heat transfer mechanism here is analogous to that at a stagnation point. The variation of the peak pressure, and hence that of the peak heat transfer, along the leading edge is due in large part to the variation of the total pressure loss through the changing nose and detached wing shocks encountered by a given attachment streamline. Proper capture of these shocks is therefore essential.

With the sublayer approximation, a streamwise (spatial) marching of the two dimensional (2D) cross-flow was possible with the PN/S method. This is to be contrasted to the ARC3D method which envolves a time-marching of a 3D flow. With the PN/S method, the mesh generation is therefore considerably simplified, and the computer time and memory requirements are significantly decreased, relative to the unsteady ARC3D method. Despite this, the above PN/S calculations required approximately 15 hours of Cray XMP computer time, clearly excessive for design purposes.

The above hypersonic solution was over-simplified from two essential aspects. The first is the assumption that the flow was locally in chemical equilibrium (infinitely fast reaction rates). This assumption is in serious error through shock waves, where the molecular vibrational energy modes and the chemical reactions are essentially frozen (zero reaction rates). Furthermore, atomic recombinations require relatively infrequent three-body collisions and are thus far from equilibrium over most of the significant part of the flow. Finite rate chemistry must therefore be incorporated into the above problem, and this requires the addition of specie continuity equations to the N/S equations. This addition will not alter the existing mixed parabolic/hyperbolic character of the equations (adding only additional degrees of parabolicity), so that a marching procedure is still permitted. When fast reactions are present however, the stiffness of the already-stiff equations is greatly increased, requiring modification of the marching procedure.

The second shortcoming in the above hypersonic solution was the assumption of turbulent flow. For the high altitude and hypervelocity conditions, the forward portion of the configuration will be immersed in laminar flow, followed by an equal stretch of transitional flow before the boundary layer becomes turbulent. This change of the boundary layer character will greatly affect the heat transfer and skin friction. Of special concern is the overshoot region occurring between the transitional and turbulent regions where the heat transfer and the skin friction can assume values significantly larger than the turbulent values. To model this transitional transport properly, a suitable prior-history differential equation model for the turbulence must be employed.

1.4. SUPERCOMPUTERS (For background, see Gentzsch [1.12])

Clearly the accomplishments to date in computational fluid dynamics would not have been possible without the workhorse vector computers as the Cray XMP and ETA 205 series computers. Such computers have achieved phenomenal power through the efficient utilization of parallel and pipeline arithmetic hardware. Crucial here was the timely transfer of data between the arithmetic units and memory with the data being transmitted, not singly, but in suitable packages (vectors) using parallel paths. The essence is to achieve a smooth flow of data through the system from the raw input stage to the finished output without congestion at avoidable bottlenecks. It is clear that the possible smoothening of the data flow depends heavily on the vector features of the hardware and the compatibility of the algorithm and its "work-breakdown" structure with these features. With the latter aspects so problem dependent, the optimal design of the vectorization cannot be relegated to the compiler or to the operating software alone. The programmer must assume a significant portion of this task himself, and it is essential that he be intimately familiar with the computer hardware to accomplish this task successfully. It is further clear, as most code users have experienced, that the vectorization tailored to one supercomputer does not automatically carry over to other computers, raising the portability issue.

1.5. SUMMARY AND CONCLUDING REMARKS

The above transonic and hypersonic examples show the high-promise of the N/S methods. However to yield viable solutions for design purposes, improvements in the methods are required to eliminate the fluid dynamic shortcomings described above. In the transonic case, a turbulence model, more appropriate than the algebraic Baldwin/Lomax model, will be required to treat the shock/boundary layer interaction correctly. One or two differential equation model will be required for this purpose; and with the shock pressure-rise being mesh dependent, such prior-history turbulence modeling must be scaled to the mesh. Additionally, to capture properly the swept forward shock and slip surfaces as the leading edge separation vortices arising on sharp-nosed swept wings, an adaptive mesh will be needed. In the hypersonic case, finite rate specie continuity equations and one or two equation, prior history, turbulence modeling for the transitional flow must be added to the existing N/S equations. The addition of these needed improvements will severely tax the already unaffordable computer cost and memory requirements. It is thus more than ever mandatory that computer programs be so constructed to be fully sympathetic to the hardware of current vector computers. Moreover, it is not too early to address the matter of portability of such codes, not only between different supercomputers, but between supercomputers and, for example, the more recent mini-supercomputers. It is the purpose of the present AGARDOGRAPH to address these issues in the following chapters:

General Considerations (Dr. Ken Neves)

Chapter 2: Hardware Architectures

Chapter 3: Algorithms and General Software Considerations

Specialization to CFD (Prof. Dr. Wolfgang Gentzsch)

Chapter 4: Vectorization of Fortran Programs at the Do-Loop Level

Chapter 5: Restructuring of Basic Linear Algebraic Algorithms

Chapter 6: Computational Fluid Dynamics and Supercomputers

A Glossary of supercomputing terms is contained in the Appendix following Chapter 6.

1.6. REFERENCES

1.1. Swanson, R., and Turkel, E., "A Multistage Time-Stepping Scheme for the Navier/Stokes Equations", NASA Contractor Report 172527, 1985.
1.2. Pulliam, T., and Steger, J., Implicit Finite Difference Simulations of Three Dimensional Compressible Flow, AIAA Journal, Vol. 18, No. 2, 1980.
1.3. Pulliam, T., and Steger, J., Recent Improvements in Efficiency, Accuracy, and Convergence for Implicit Approximate Factorization Algorithms, AIAA Paper No. 85-0360.
1.4. MacCormack, R., A Numerical Method of Solving the Equations of Compressible Viscous Flow, AIAA Paper No. 81-0110.
1.5. Thomas, J., and Walters, R., Upwind Relaxation Algorithms for the Navier/Stokes Equations, AIAA Paper No. 85-1501.
1.6. MacCormack, R., Current Status of Numerical Solutions of the Navier/Stokes Equations, AIAA Paper No. 85-0032.
1.7. Ying, S., Steger, J., Schiff, L., and Baganoff, D., Numerical Simulation of Unsteady Viscous High Angle of Attack Flows Using a Partially Flux-Split Algorithm, AIAA Paper No. 86-2179.
1.8. Kaynak, U., Holst, T., and Cantwell, B., Computation of Transonic Separated Wing Flows Using an Euler/Navier-Stokes Zonal Approach, NASA TM No. 88311, 1986.
1.9. Johnson, D., "Predictions of Transonic Separated Flow with an Eddy-Viscosity/Reynolds-Shear-Stress Closure Model", AIAA J., Dec. 1986.
1.10. Karman, S., Steinbrenner, J., and Kisielewski, K., Analysis of the F-16 Flow Field by a Block Grid Euler Approach, Presented at the AGARD meeting on "Applications of Computational Fluid Dynamics in Aeronautics, Aix en Provence, April 1986.
1.11. Blom, G., Wai, J., and Yoshihara, H., Hypersonic PN/S Calculations over a Generic Wing/Fuselage, Boeing Report BMAC Aero TN 111A, 1986. (Also presented at the 1986 SAE Aerospace Technology Conference, Long Beach, CA, 1986.)
1.12. Gentzsch, W., "Vectorization of Computer Programs with Application to Computational Fluid Dynamics", Notes on Numerical Fluid Mechanics, Vol. 8, Vieweg, 1984.

VISCOUS VORTICAL FLOWS

PROBLEM 1:

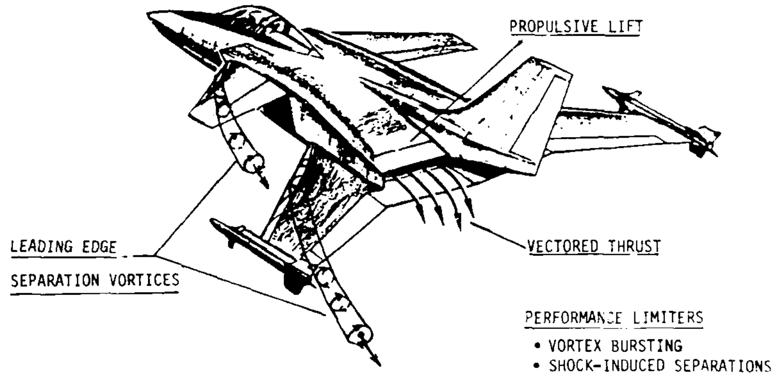


Figure 1. Transonic Maneuverability of Combat Aircraft.

PROBLEM 2:

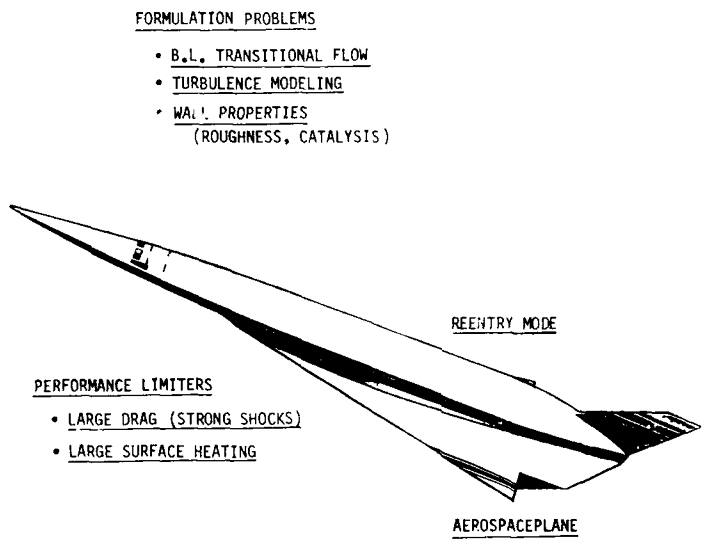


Figure 2. Hypersonic Cruise and Reentry Performance.

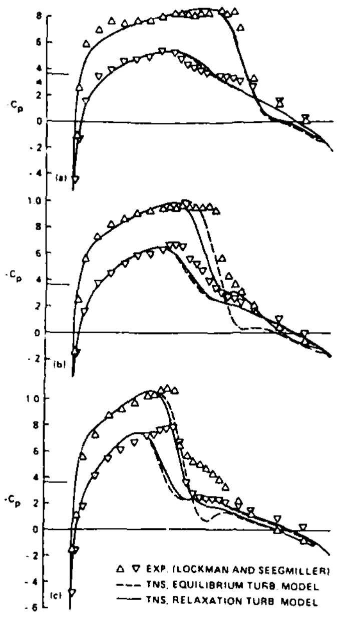


FIGURE 3 . Comparison of Pressure Distributions – 20°-Sweep Untapered Wing.

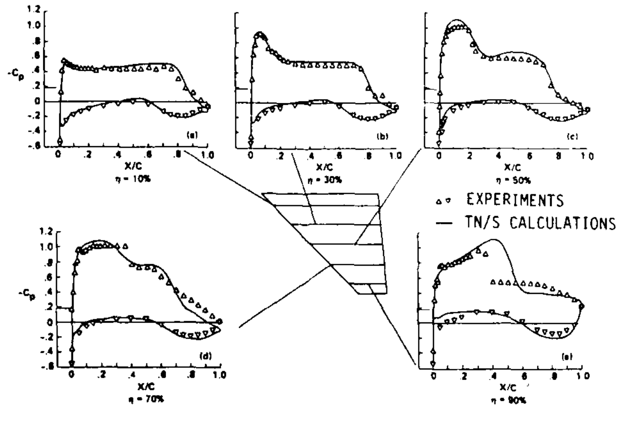


Figure 4 . Test/Theory Comparison of the Pressure Distributions for a 45°-Sweep Wing.

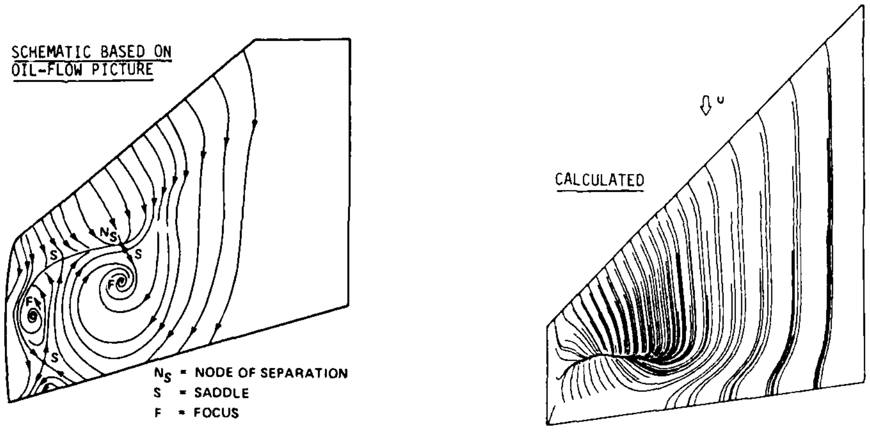

SCHEMATIC BASED ON OIL-FLOW PICTURE
CALCULATED
N_S = NODE OF SEPARATION
S = SADDLE
F = FOCUS
S
F

Figure 5. Comparison of the Skin Friction Lines (45°-Sweep Wing).

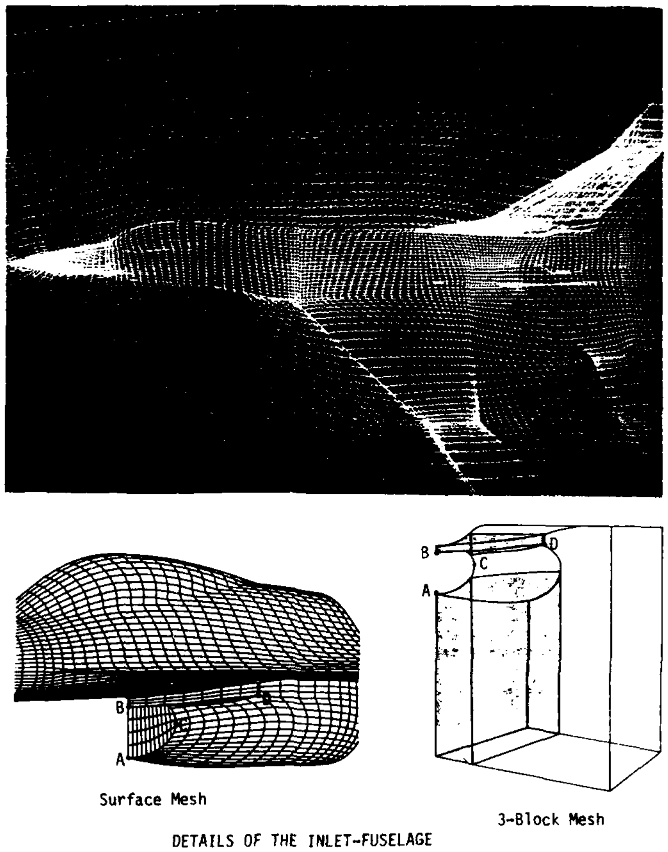


Figure 6. Mesh for the F-16 (Ref. 10).

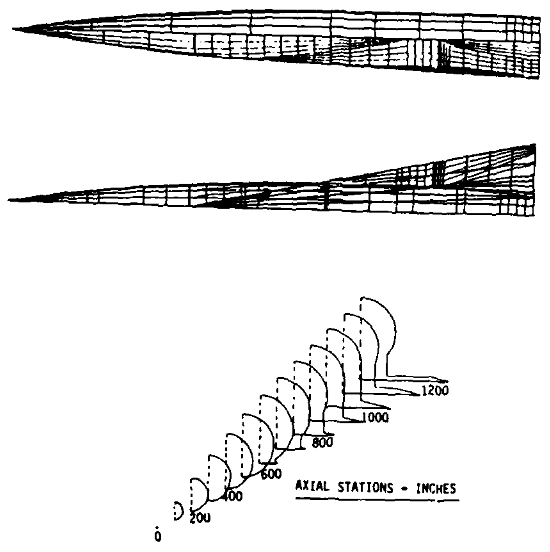


Figure 7. Generic Hypersonic Wing/Fuselage Configuration.

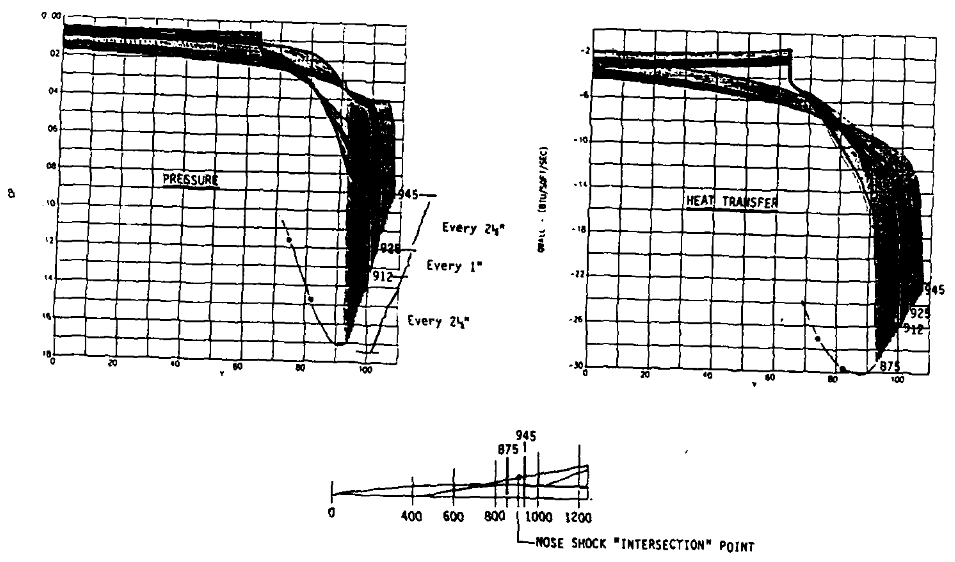


Figure 8. Spanwise Variation of Pressure and Heat Transfer (Generic Configuration).

PREFACE TO CHAPTERS 2 AND 3 (Dr. K. W. Neves)

The purpose of this treatise is to explore the technology employed in today's supercomputer designs. The author has assumed the reader is not generally aware of this technology. To help in the reading of this document, a glossary has been provided for terminology peculiar to supercomputing. At times more common computer jargon is used, but for the most part, it is explained when used. The technology discussed is a mixture of hardware technology, computer design, and algorithm methodology. The concepts presented are often summarized in sections entitled "important concepts." The focus of these summary remarks is to attempt to isolate more lasting principles from the particular machine designs. Thus, it is hoped that the rapidly changing characteristics of commercially available computers have been separated from longer lasting trends. As a result, in-depth comparisons of machines were avoided. Upon reading this work one should not expect to discover which of the 5 supercomputers, 4 minisupercomputers, and assorted parallel computers discussed, is superior. By the same token, to illustrate particular technical points, machine comparisons are frequently used that do involve the various machines. Weaknesses, bottlenecks, and strong points of each machine have been uncovered through a series of discussions which lead to some exposure of the dependence of algorithm design on computer architecture.

This treatise is particularly timely in the author's estimation due to the fact that the computer industry is rapidly approaching another milestone in high-end computer hardware. Roughly 15 years ago, the fastest computers made (notably CDC followed by Cray Research) used a new design method -- vector pipelined architecture. It is now almost universally accepted that advanced hardware today uses this approach to achieve superior price performance. The milestone we are now beginning to observe in this class of computers (i.e. supercomputers and minisupercomputers) is that of the combined use of pipelining and CPU-parallelism. Neither pipelining nor parallelism is new. But their use in computers aimed at the general purpose scientific market has caused a tremendous challenge to software professionals responsible for the computations that fuel technology and research in science and engineering. For these reasons this treatise has attempted to accomplish two things: 1) explore today's approaches to vector computing by relating hardware architecture to software/algorithm design, and 2) give some insight into potential trends in parallel computing that are likely to be commonly used in supercomputer design over the next five or so years.

Chapter 2 is largely devoted to hardware considerations. The design of three Japanese and three American companies are reviewed and contrasted -- but always with an eye toward computational strategy (i.e. algorithm and software ramifications.) Chapter 3 is devoted to software issues such as algorithm design, transportability, software migration, and benchmarking methodology. One principle is frequently supported throughout this treatise, while it is never stated. It is simply that there is no elementary formula for optimal usage of today's supercomputers. Software performance can only be optimized by those who have a curious mixture of understanding of both the hardware architecture and the application and its underlying algorithms. The automation of program optimization is not necessarily an impossible task, yet it does not seem to be able to accommodate the rapidly changing issues in this ever expanding technology.

ACKNOWLEDGMENTS: Chapters 2 and 3 are the result of a broad collaboration spanning many years. It reflects the insight and knowledge the author has gained with a long association of computer professionals at the Engineering Technology Applications Division (ETAD) of Boeing Computer Services. Among the many individuals in this category several have had continual impact with insightful comments and discussion over the years. While the list could be very long, the author would like to especially acknowledge the following: B. Dembart, M. Epton, A. Erisman, R. Grimes, J. Lewis, and H. Simon. In addition, this material benefited from the author's work responsibilities in directing the High Speed Computing project. It was under the auspices of this project that a great deal of information was gained during a project trip to Japan in 1986 with B. Wilson and J. Kowalik, also with Boeing Computer Services. The author would also like to express appreciation to J. Kowalik for his thoughtful reading of of Chapters 2 and 3.

CHAPTER 2: HARDWARE ARCHITECTURE (Dr. K. W. Neves)

2.1 BACKGROUND

In this chapter, an overview of computer architectures is given. The purpose, however, is not to offer a text book description of the electronics behind computing, but rather an appreciation of the barriers in computer development that have led to the comp'ex design of today's supercomputers. This introductory section will examine 1) the need for increased computer power as result of the pressure from new computational processes; 2) the limits and barriers of hardware; 3) a brief history of parallelism; and 4) an overview of parallelism in today's computers. Following this Background section, an in-depth look at today's supercomputer architectures and emerging computer products is given. In chapter 3, the impact of new hardware architecture on algorithms will be examined.

Like computational fluid dynamics, supercomputing has developed a set of terms, definitions and ad hoc descriptions of common processes. In this monograph, most of these terms will be defined when used. Nevertheless, in an attempt to aid the reader an informal glossary of supercomputing terms is included in an appendix.

2.1.1 The Push for More Computational Power

To appreciate the need for innovation in computer hardware architecture, it is appropriate to appreciate the need for more computer power. Modern engineers, scientists, and researchers are spending an ever greater percentage of their efforts on computers. Ken Wilson, Nobel Laureate in Physics, ascribes the birth of a new branch of physics to the increased capacity of today's powerful computers [2-1]. No longer is the experimental physicist confined to the laboratory, and no longer is the theoretical physicist tied solely to his creativity. Experimental computation, the melding of experiment and theory, has been made possible through the dynamic growth in computer power. Electronic computation has become an indispensable tool for all of science. For example, a decade ago the fundamental tool to verify and/or repair a wing design, was the wind tunnel. Clearly, this is an indispensable tool. Even in an ideal wind tunnel experiment, at best, one cannot even accurately measure the total drag. To differentiate the components of measured drag is difficult, if not impossible. Computationally, one can separate the effects, analyze them and experimentally find potential solutions by isolating physical parameters which in live experiments are inaccessible. It is also possible to "experiment" computationally in areas where physical experiments cannot be done, e.g. in other atmospheres or ultra high temperatures. Thus, simply as a tool of science, computers are being used more and trusted for functions that formerly were the domain of real experiments or pure theory.

Increased usage of computer tools can account for "more" computers, but not necessarily account for the seemingly unending need for more powerful computers. How can today's already complex computations require more and more computer power? The answer can only be revealed by examining trends in both the use and design of computers themselves. There are basically four pressures for more computational capacity in computing hardware:

1) More refined analysis,

2) more demanding models,

3) new user interface requirements, and

4) design and optimization of complex processes.

The first pressure for more computational capacity comes from the requirements of more detail and accuracy in established models. With no changes in the model or computer implementation of the algorithms, more computer power can still be required to perform the desired studies of processes like computational fluid dynamics. Most physical models are discretized into 2 or more dimensions. This is quite often done by grid-point representation of what was originally two or three dimensional curves or surfaces. The grid spacing often is directly related to the fidelity of the model to the "real world". Thus, finer grid spacing results in a more accurate and more meaningful result. In two dimensions, simply refining the grid by halving the grid spacing leads to a 4-fold increase in data and computation. Moving the analysis into four dimensions (three space and one time dimensions) can call for another 16-fold increase in required computational power in order to double accuracy. Table 2-1 below (taken from reference [2-2]) illustrates the increase in complexity for several methods in 2 and 3 dimensions.

COMPLEXITY IN FLO 57

ALGORITHM	FLOPS/CELL /CYCLE	NO. OF CELLS	NO. OF CYCLES	TOTAL OPS. X 10**9
POTENTIAL (3-D)	500	10,000	100 - 200	5 - 10
EULER (2-D)	400	5,000	500 -1000	1 - 2
EULER (3-D)	950	100,000	200 - 500	20 - 50

(Note: cycles refer to algorithm iterations to attain steady state accuracy for each cell)

TABLE 2-1

Many of today's production and research programs in computational fluid dynamics are limited by the computational bottlenecks of computational speed and memory capacity, even on today's fastest supercomputer systems.

The table above also illustrates the second pressure for more computational power, that of more refined analysis. This was discussed at some length in the introductory chapter relative to CFD. Depending on the physical model being modeled more refined computational models are required. One often finds compromises in models or the objects being modeled to compensate for the lack of computational power. The following diagram (also from reference [2-2]) illustrates the compromise process.

```
                      METHODS                              GEOMETRY
                                                      (COMMONLY FEASIBLE)

             C                                                 G  C
             O    LINEAR INVISCID            FULL AIR FRAME    E  O    /\
  ||     M   M    NON-LINEAR INVISCID                          O  M    ||
  ||     O   P    TRANSONIC POTENTIAL        3-D WING          M  P    ||
  ||     D   L    EULER                                        E  L    ||
  ||     E   E    REYNOLDS-AVERAGED N/S      2-D AIRFOIL       T  E    ||
  ||     L   X    LARGE EDDY SIMULATION                        R  X    ||
  ||         I                                                 I  I    ||
  ||         T                                                 C  T    ||
  \/         Y                                                    Y
```

Figure 2-1
Complexity and Compromise

Of course, the situation is even more complicated. As discussed in the introduction, the model itself can become more complex depending on the phenomenon being studied. For example, angle of attack, mach number, viscosity, vortices and the like all impact the model, the mathematical equations, the algorithms, and hence, the efficacy of the computer design.

Strangely enough, another factor in demand for computational power comes from the complexity of the process itself. The ability to compute more refined meshes on more refined models has developed more sophisticated interaction between the scientist and the computer itself. An example, of this is apparent in commercial airplane design. One of the important factors in airplane design is the interaction between wing design, pylon interference, and lift. In order to observe anomalous behavior it has been common to compute and "observe" pressure gradients on the wing. It is interesting to trace this "observation" process. Not too many years ago the pressure was analyzed at various wing cross sections. Today, it is not uncommon to see color coded pressure gradient profiles along the entire wing. The trend is, however, toward observing the (particle injected) flow over the wing in real time interaction with varying angle of attack. The more the modeler can "see" in this type of process, the greater the demand for model refinement and computational power for near real time display. Processes that were one or more CPU hours a few years ago, are now required to be performed in milliseconds in order to achieve this type of interaction.

Finally, the most overwhelming need for increased computer capability comes from the trend toward optimal design. Many of today's most complex processes are direct analysis of physical behavior. The results are fed back to the engineer or scientist, who in turn defines a new set of inputs or model changes to analyze further. The trend toward taking the man "out-of-the-loop" is very evident today. To illustrate this point, consider a simple two-dimensional airfoil. One can parameterize this curve and perform an analysis of pressure as a function of flow over the airfoil. Figure 2-2 displays the pressure profile plotted against the chord-wise station. The dotted lines correspond to the dotted airfoil displayed in Figure 2-3. These results are the output of an analysis program that simply analyzes pressure given the airfoil as input.

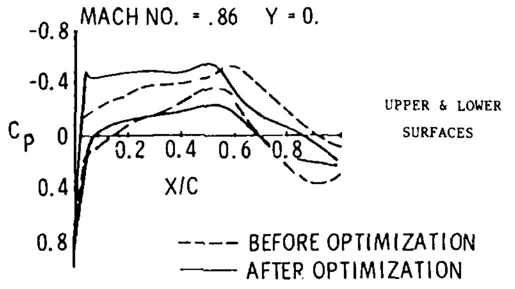


Fig. 2.2
Pressure Over Wing: Two Cases

If you consider several parameters defining the curve as the domain of a function space, and parameterize the output of the computational process to form a range space, a design process can be defined. An objective function can be defined to, say, maximize the area under the pressure curve. Using standard optimization techniques with the analysis program as the "function box" a simple optimization problem is defined and can be solved. The result is displayed in Figure 2.2 with the solid lines, now as input to a design process. The result is an improved airfoil, depicted by the solid shape in Figure 2.3. This not only has been done in this simple case, but is being investigated by researchers in much more complex situations and in other industries on completely different types of problems. The point to be remembered is that "today's analysis programs are tomorrow's inner-loops." A word of caution is worthwhile. The process of creating design methods from analysis tools is not trivial. Quite often the objectives are not easily defined and may reside in the "gut feel" of experts. In addition, first attempts often lead to unrealistic results, and require pains taking refinements. Nevertheless, this trend is continuing to show promise in many disciplines.

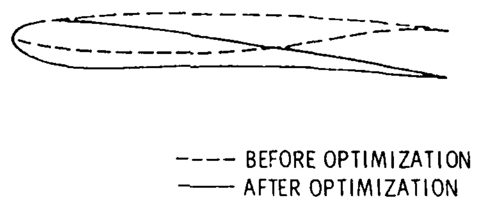


Fig. 2.3
Optimal Airfoil

While this has not been an attempt to give a comprehensive survey of the technological demand for more computer power, it is hoped that the need for such power is apparent. In fact, the advent of new approaches to the use of computers in fields such as artificial intelligence, vision, chemistry and robotics suggest the only potential abatement in the quest for increased computer power would be a sudden end to human imagination and invention!

2.1.2 The Hardware Barrier

In the course of technological history, electronic computation is a very recent event. In its short evolution, the basic design of computers changed very little. Advancement in computer power came largely from improvements in electronic circuitry. The big breakthroughs were electronic tubes, transistors, chips, etc. Each new invention seemed to circumvent a previous hardware problem. As the speed of computer circuits has increased, however, some real "hard" limits are becoming evident. These hardware realities are being addressed by the computer architect. The architect is caught in the middle of hardware realities and the demand for increased computational power described in the previous section.

The biggest "hard limit" faced is the speed of light. A modern supercomputer has cycle times on the order of nanoseconds (currently, from 4 to 20 nanoseconds depending on the company.) The industry is pushing steadily toward the goal of 1 nanosecond. In one nanosecond, an electron travels about a five centimeters. The result is that even with infinitely fast gate and switching speeds, chip interconnection delays will prevent computer cycle times from going to zero. In today's computer designs roughly 60% of the cycle time is due to switching speeds, 25% chip interconnections, and 15% design compromise. With this reality the computer architect has been increasingly willing to dabble with parallelism, pipelining, overlapped processes and the like (to be described later.)

A complete description of the history of electronics and chip technology is not only beyond the scope of this treatise, but beyond the expertise of the authors. However, it is important to understand the difference in the "building materials" from which computer architects may select. An understanding of the trade-offs and compromises that various architects must make, can lead to a better understanding of the resulting hardware, and surprisingly, a better understanding of appropriate software design.

Materials and components found in modern supercomputers, of course, vary. The variation is not only in material used but in the degree of integration (circuits on a single chip) and methods for packaging and cooling. In fact, most supercomputers use ECL (bipolar emitter coupled logic) chips for logic. Even IBM has used ECL in the 3090, a departure from their tradition. (The notable exception is the ETA Systems GF-10, which uses complementary metal oxide semiconductor, CMOS, technology.) MOS (metal oxide semiconductor) integrated circuits (ICs) are being more frequently substituted for ECL, particularly in memory. The use of MOS as opposed to ECL is a very good example of the type of compromises faced by the architect. MOS, for example, has lower power requirements, hence less heat to dissipate; however, MOS circuits are slower and add to the "wait" time when used in memory. Thus, the choices begin to become evident -- larger, but slower memory versus smaller, yet faster, memory. The choices are not quite that simple. For example, while MOS may mean slower fetch time for a single piece of data, large amounts of data can be streamed (or pipelined) in such a manner as to still offer high bandwidth data retrieval. We shall discuss the impact of these types of decisions on software in a later chapter. CMOS (complementary metal oxide semiconductor) circuits are being considered by manufacturers. As mentioned the ETA GF-10 is using these chips. Their advantage comes from very low power requirements leading to high density packaging which reduces the interconnection distances through very large-scale integration. Their slower speeds are the problem. ETA has chosen to cool them to 100 degrees Kelvin to improve speed.

At first blush, one might conclude that supercomputers are not all that different. After all, they mostly use ECL for logic and MOS for memory. However, due to packaging and degree of integration the resulting hardware is very different. This becomes most apparent when looking at heating/cooling characteristics. The table below illustrates this fact.

Cooling Technology

COMPUTER	COOLING TECHNOLOGY
FUJITSU VP-200 (AMDAHL)	AIR COOLED
HITACHI S-810/20	AIR/MEMORY & WATER/LOGIC
CRAY X-MP	FREON
CRAY-2	LIQUID IMMERSION
NEC SX-2	WATER
ETA GF-10	LQD. NITROGEN IMMERSION/LOGIC CHILLED AIR/MEMORY
IBM 3090/VF	WATER

TABLE 2-2

The future holds promise for alternate materials. Cray Research has already announced that its CRAY-3 will be based on galium arsenide. Current projections by industry analysts, however, are that successful galium arsenide chips are likely to produce cycle times on the order of 1-2 nanoseconds, and that more exotic technology such as HEMT (high electron mobility transistors) are required to achieve significant improvement in cycle times. Today almost half the cycle time is due to packaging. New materials promise improvement in gate delays (picoseconds vs. nanoseconds in logic ICs), less heat dissipation, and faster saturated electron speed over Silicon. However it is necessary to employ larger scale integration to achieve significant improvements. VLSI (very large-scale integration) is characterized by 1000's of transistors/IC. CRAY-3 will have 11,000 transistors per IC. The predictions are for millions of transistors per IC by 1990 (VHSIC, very high-scale IC). With this kind of integration the need for larger numbers of chip interconnections is reduced, thereby "side-stepping" the speed of light issue. The complexity of the circuits however, present there own dilemma. How can all these logic circuits be effectively used? Once again, the answer will have its impact on software and applications.

Most architects seem to be mindful to some degree of the impact of their decisions on the user. Nevertheless, the successful architect seems to be a wily mixture of engineer, scientist and magician. Unfortunately, the genius of a good architect is just the beginning of a usable machine. The ever increasing complexity of the computer parts (chips) leads to ever increasing difficulty in using the full potential of the hardware. In the next subsection a very brief discussion of the use of parallelism in computer architecture is provided. Parallelism is not a new concept. As the complexity of the computer and its underlying building materials has grown, it is natural to turn to concurrency in software, systems, and hardware to maximize yield for a given level of hardware technology.

2.1.3 Parallelism in Computing

Long detailed taxonomies of computing architectures are not only boring, they often add little to the understanding of the software implications of various architectural approaches. For example, a long discourse as to "what is a supercomputer" or whether the machine is parallel, vector or single/multiple instruction streamed etc., contributes little to the fundamental issue of usage methodology or prediction of performance. By the same token, these same characteristics figure greatly in software design for these unusual systems. As computer architects use more (and often recurring) tricks to push performance beyond what component technology alone can offer, a greater obligation is placed on the users to accommodate these changes. The basic tool the architect uses to achieve "super" performance is parallelism. This parallelism comes in many forms. In this section we will review general forms of parallelism used in high performance computing without attempting to classify vendor specific products.

In Section 2.2 we will examine architectural features of supercomputers in greater detail, particularly those features that greatly impact computation. In Section 2.3 a brief overview of minisupercomputers (to be defined) will be given, along with commercial examples of purely parallel systems.

Historical Perspective: The term parallelism is so often used that it almost has taken on specialized meanings in various circles. A popular concept concerning parallel computers is that they are the class of machines that have many copies of a system (CPU) working simultaneously. This, indeed, is a form of parallelism. Yet, from a broader perspective, parallelism is a term that may be more precisely replaced by "concurrency", the simultaneous operation of one or more hardware functions. In this sense, parallelism has been with us from the very early days of computing. For example, the overlap of computation and I/O is a form of parallelism. In modern supercomputers, the parallelism has invaded the very heart of computation -- the floating point operation itself. This latter phenomenon will be discussed at some length, but first, a historical perspective of the use of parallelism at the system level is in order.

A thorough, yet concise, history of parallelism is given in Hockney & Jesshope [2-4]. They indicate that parallelism has been used and considered by computer designers from the earliest days as indicated from the following excerpt:

> The earliest reference to parallelism in computer design is thought to be in Gereal L. F. Menabrea's publication in . . . October 1842, entitled Sketch of the Analytical Engine Invented by Charles Babbage. . . In listing the utility of the analytic engine, he writes: ". . . Likewise, when a long series of identical computations is to be performed, . . . the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."

The following table of historical usage of parallelism in computer design was assembled from several sources, most notably [2-4] and [2-5]. Where computers are named, they are merely examples of commercial products that have used the feature, not an indication that the company listed necessarily was the first to use the particular form of parallelism. It is evident that the occurrence of concurrent operation of I/O, memory fetch/store, functional units, and/or instructions has been a mainstay of computer design from the inception of the industry.

Examples of Parallelism in Computing

USE	COMPUTER	DATE
Bit parallel computation	IBM 701	1953
I/O peripherals (simultaneous I/O)	IBM 709	1958
Simultaneous use of the address unit and the arithmetic unit, and pipelined operations	ATLAS	1961
Overlapped, look-ahead instruction decoding	STRETCH	1961
Parallel CPUs	SOLOMON (ILLIAC)	1974
Functional parallelism, banked memory	CDC 6600	1964
Functional parallelism of pipelined units	CDC 7600	1969
Associative memory	GOODYR. STARAN	1972
Vector instructions	CDC STAR 100 TI ASC	1973 1973
Multiple vector pipes	TI ASC	1973
Array processors	FPS AP-120	1976
Vector registers	CRAY-1	1976
Pipelined instructions (MIMD implementation)	DENELCOR HEP	1982
Multiple vector CPUs	CRAY X-MP	1982

TABLE 2-3

Forms of Parallelism: The topic of parallelism in computers could easily require a monograph of its own. This discussion will be confined to aspects of parallelism in modern computers that have a direct impact on scientific software design. To offer an example, consider multiple central processing units (CPUs). Several computer manufacturers offer systems with multiple CPUs. Notably, IBM and their compatible competitors have had such machines for years. The impact of the introduction of multiple CPUs on software design, however, has been negligible because their use has been primarily to improve throughput within the job stream not within a single job. In 1982-3 CRI began to offer "multitasking" of a single job on multiple-CPU CRAY X-MPs. More recently, IBM has offered operating systems enhancements that allow the single Fortran program the ability to utilize multiple CPUs in the 3090/400. This indeed has software design implications.

Before discussing forms of parallelism in more detail, several commonly used terms employed in discussions of parallelism need to be defined. The terms are from a classification scheme or taxonomy of computers due to Flynn [2-6]. The purpose of defining them here, is not for classification, but for their utility in describing potential uses of given computer designs. The terms represent computer processes for performing instructions and the flow of data resulting from these instructions. They are defined as follows:

SISD (single instruction stream/single data stream) - this is the conventional approach to computers, today often called scalar computing (which is slightly inaccurate, except that most scalar computers are also SISD). The concept is that instructions are processed sequentially and result in a flow of data from memory to functional units and back to memory (perhaps through caches or registers). The data flow is equally sequential. This does not exclude pipelining in the process, but does exclude manipulation of vector data types explicitly.

SIMD (single instruction stream/multiple data stream) - the processing of instructions remains the same, but the data manipulated can be explicitly vector data. In parallel machines the data could come from separate memories simultaneously. In vector machines the process could be argued as still being SISD with the instructions simply being macros (microcoded subprograms) launching SISD processes. This latter interpretation has been dropped since the process of vector computers is more similar in nature to true parallel SIMD from a software design perspective. Each "vector" instruction is viewed as manipulating an array of data stored in single or multiple memories according to the architectural design.

MIMD (multiple instruction stream/multiple data stream) - this process implies the ability to simultaneously process (decode) instructions. These instructions are then free to process independent data streams.

While the above are distinct processes, the architectures that implement them can be quite varied and, in some cases, support more than one of the above. The three techniques of parallelism in hardware design seem to be most common:

1. Simple replication--Identical and multiple operation of hardware functional units and/or CPUs.

2. Multiple non-identical functional units.

3. Pipelining -- the segmentation of a single functional unit or process to achieve assembly line type operation leading to the simultaneous overlap of independent inputs.

It is interesting to note, that it is rather difficult to categorize machines by the above definitions, and even if one could, the implications of each class is not very distinctive in terms of its impact on software.

Combinations of these parallel hardware design techniques can be, and are, employed in subtle ways. Perhaps the more obvious uses of these techniques are related to the floating point computation process in modern computers. In fact, all of these techniques have been used in the processing of instructions by various computers over the years. It is quite likely that computers of the future will increase the use of these techniques in all three areas of computer design: computation (functional units), instruction processing (instruction or control units), and data movement and handling (memory and I/O).

The simplest of technique is replication. Yet, its simplicity becomes obscure when considered against the variations in its implementation. For example, consider employing replication in floating point computation. Immediately, there are two choices. One can design the CPU to include more than one functional unit, such as a floating point adder. One could employ any number. This of course has fundamental software implications beginning with the compiler (usually Fortran) which does not have any concrete way to restructure software to insure that two or more adders can be kept busy. Furthermore, there is an issue related to "hardware balance." Increasing computational power without a corresponding increase in bandwidth to data memory can create a serious bottleneck that could easily nullify the value of increased functional units. Another way to use replication to improve floating point performance is to simply add more CPUs. This not only increases the number of floating point units, but increases the number of instruction processors. This creates further decisions to be made about data flow. In this situation the Fortran compiler is helpless. The compiler must be extended to handle parallel constructs and/or provide tools for direct user intervention. In this setting, questions arise related to memory contention, multiple memory units dedicated to each new CPU. (In Section 2.3 several parallel architectures will be discussed and contrasted.) As mentioned before, multiple CPUs can be used solely to increase throughput in the job stream and have little effect on individual jobs, or mechanisms can be added to the operating system and programming language to allow the multiple CPUs to be used on one computation. The latter approach has considerable impact on software, and the payoff is a "delicate" function of the number of CPUs provided. From a single job perspective the latter approach can be an MIMD process, and the former a SISD or SIMD process. In fact, a successful computer architect, employing replicated parallelism, invariably must make strategic decisions related to the ultimate use of the new design -- at the very least it must be decided whether the primary goal of user interaction will be via SISD, SIMD, or MIMD processes.

The second approach to parallelism is only a refinement of simple replication. Instead of the addition of identical units, add functional units that are independent, and perhaps different. This could be accomplished in many ways. The overlap of instruction address computation with the floating point units is a common instance of this type of parallelism. Another recent example is the addition of floating point accelerators and matrix multipliers in various designs. These are simply added to the computers CPU. The "parallelism" comes from the concurrent operation of these units as independent functional units. Perhaps the most pervasive use of this technique is in the Floating Point Systems array processor. The system is composed of 10 to 15 functional units including various types of data storage and registers (data pads). The instruction word is wide enough so that in a single instruction as many as 13 simultaneous instructions can be directed independently. (Viewed in this manner one would classify the whole FPS line as MIMD. In fact, most computer taxonomies have difficulty classifying the FPS architecture.)

Probably the most used form of parallelism in modern scientific computers, is pipelining. The analogy of the automobile assembly line is often applied to this process. In Section 2.2, specific implementations of pipelined arithmetic units will be examined in the context of the total CPU architecture. Here, as an introduction, a brief description of a pipelined floating point addition unit is given.

Typically, high performance computers are at the peak of hardware chip gate and interconnect speeds for a given price performance. The cycle time, or clock period is the predesigned synchronization period for the various simultaneous functions that go on in a CPU. Most of these functions are completed within a single cycle. Floating point operations, in general, are among the most complex operations and are typically executed in four or five machine cycles. (See Figure 2-4) Pipelining the operation of a functional unit, in today's supercomputer technology, often leads to a functional unit that requires 7 or 8 cycles to produce a single result.

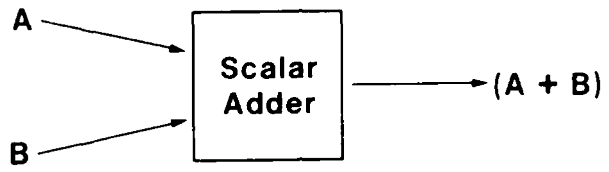


Figure 2-4
Scalar Computation

At first glance, this is not a very good trade-off. In fact, pipelined units are much slower in producing a single result. Their utility is only realized through both software and hardware instruction processing changes. This will be discussed at length in Section 2.3. In order to describe the pipeline process, consider Figure 2-5. Assuming that a steady "stream" of data is provided to the pipelined unit, an assembly line processing can be implemented using a "segmentation" of the task. Figure 2-5 illustrates the result of providing 8 operand pairs to the pipeline in Figure 2-6 after the process is eight cycles in to production.

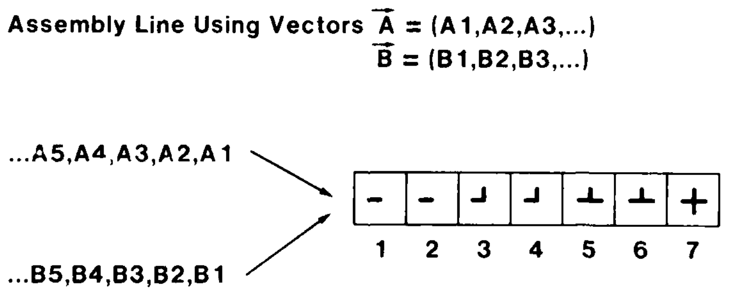

Assembly Line Using Vectors $\vec{A}$ = (A1,A2,A3,...)
$\vec{B}$ = (B1,B2,B3,...)
...A5,A4,A3,A2,A1
...B5,B4,B3,B2,B1
1 2 3 4 5 6 7

Figure 2-5
Pipelined Arithmetic

The result is, that at the end of eight cycles, each successive operand pair is in some stage of completion. If the "stream" is long enough, every cycle after the eight produces a result. The asymptotic result is an operation every cycle rather than one every three or four cycles as in Figure 2-4. This process has been exploited in many computer designs.

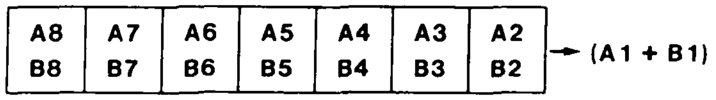


Figure 2-6
Pipelined Arithmetic: A Snapshot

It is important to note that pipelined processes often can be successively combined to yield even more effective results. This process has been named differently by various manufacturers. "Chaining" or "linking" are the most common terms used for this successive combination process. The process is one of taking the results of one pipelined process and streaming them into another. Figure 2-7 illustrates the "chaining" of an add and multiply. The most common use of chaining occurs in memory fetch/store pipelines being chained into functional units through buffers or registers.

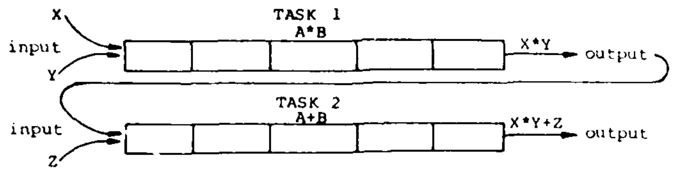


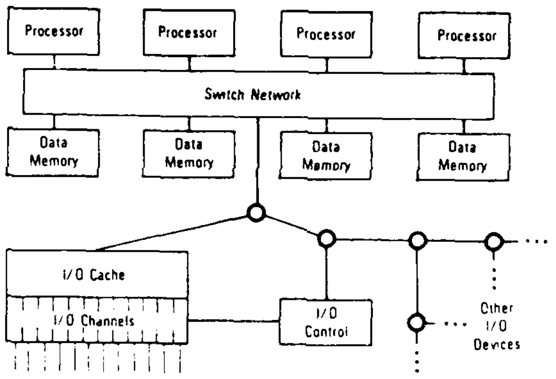
Figure 2-7
Chaining

In the case of today's supercomputer pipelining has been implemented with an SIMD process producing enormous gains in computer performance. This introductory discussion of pipelining will be amplified and refined in various settings throughout this chapter. While we have examined the pipelining of a functional add unit, examples of pipelined instruction processors, memory fetch/store, and even pipelined usage of parallel CPUs exist in today's commercially available computer designs. A unique example of architectural ingenuity employing pipelining is given in the next subsection.

2.1.4 An Unusual Example of Computer Architecture: The HEP

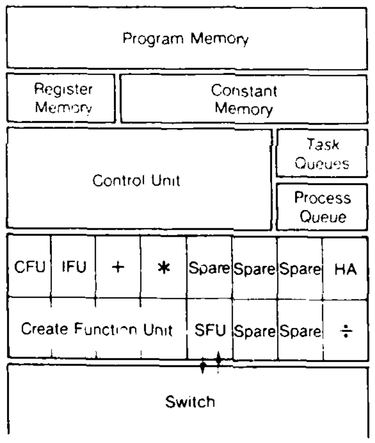
In the previous sections we have discussed three forms of parallelism (CPU replication, multiple functional units, and pipelining). In addition, three types of processes (SISD, SIMD, and MIMD) were described. As will be observed in the next section, the term supercomputer has been almost universally applied to the class of pipelined machines that employed SIMD processing. In fact, the difference between the pipelined, yet scalar, CDC 7600 and the pipelined, but vector oriented, CRAY-1 is primarily the expansion of hardware philosophy to create a SIMD process. In this section, using the Denelcor Heterogeneous Element Processor (HEP), many of the concepts of the previous sections will be illustrated in unusual ways, for this is an unusual design. We will not debate the merits or disadvantages of this computer, but explore the unusual application of pipelining, replicated parallelism, SIMD, and MIMD processes that result. For a more in-depth look at the HEP, see [2-13].

The heart of the HEP is the central processing unit called *Process Execution Modules* or PEM. The design can accommodate up to 16 PEMs. Figure 2-8 illustrates a classic approach to simple parallelism through replicated CPUs and data memory. In this approach the memory modules are accessible through a switching network. (Section 2.3 will discuss other approaches in multiple CPU architecture.)



Multi-PEM, HEP System
Figure 2-8

The HEP design supports full MIMD processing. Coordination and synchronization is accomplished through simple extensions to Fortran allowing memory based task synchronization. The system can also support both independent job streams and the logical equivalent of more classical SIMD lock step processing.



Denelcor HEP, a Single PEM
Figure 2-9

The more unusual application of parallelism and pipelining occurs within the PEM itself. Each PEM can support up to 8 instruction streams concurrently. Figure 2-9 displays the hardware diagram of a single PEM. One normally would expect 8 instruction streams to require 8 instruction processors. The HEP approach was to apply the pipeline concept to the instruction processor (IP). This is, however, a simplification. In reality the entire CPU is replicated through pipelining. The system segmentation is implemented in 8 segments. The instruction processor, the functional units such as the add/multiply, and the memory fetch/store units have this level of segmentation. The result of this strategy is to "hide" the hardware latency of the pipelining process. The latency is hidden in the sense that for a given instruction stream, instructions issue every eight cycles. Therefore, the instruction stream cycle time is effectively eight cycles--thus, hiding the latency. To be more precise, the instruction processing is split into eight steps in pipeline fashion, so that at any given instance as many as 8 instructions are in some stage of execution. The pipelining is effectively "chained" throughout the system among all the functional units. As an example, if 8 successive add instructions were issued to a PEM (assuming no memory or other resource contention is encountered), the PEM would be processing 8 successive additions in both the instruction and functional unit pipes. This would be the beginning of an SIMD add. Logically, the system has more flexibility than simple SIMD processing. The eight instructions can be associated with 8 separate processes. In fact, treating the PEM as logically 8 separate CPUs is quite often the way in which the HEP was described by Denelcor. The 8 logical CPUs could support up to 50 parallel processes. The synchronization of these processes on the HEP was done through very simple extensions to Fortran. Since all processes shared one common physical memory, a simple set of semiphore variables were added to the Fortran language. This allows memory locations to be open or closed by each process creating the ability to synchronize according to the state of certain memory variables. Processes trying to access unavailable data or other functional units are held in a logical spin state until the required resources become available. It is not uncommon for parallel computers to have simple mechanisms for synchronization and parallelization. This, however, does not necessarily mean that common algorithms or applications are easily or naturally parallelized.

The HEP is no longer made, and never seriously contended with the modern supercomputers of its day (largely due to its slow circuit speeds and the economic realities of a small company -- that is, the ability to finance the necessary commercial upgrade required to be competitive in a timely fashion.) Nevertheless, the unique application of pipelining, parallelism, and data memory access showed an amazing degree of innovation. Other unusual applications of parallelism in instruction processing are anticipated over the next few years. Architectures with large parallel instruction words are already in design. Data flow machines which treat the instructions as the entities to be parallelized, as well as the the data, are also available.

2.1.5 Advanced Concepts in Algorithm Pipelining

In the previous sections we have discussed hardware techniques to increase the concurrency (and hence speed) in computation. The techniques often used involve replication, multiple functional units, and pipelining. Given that these are used individually or in combination in a hardware design, a natural extension of their performance improving power can be employed through a process called "algorithm pipelining." This is not a standard term, in fact, it is often not described in treatises of this kind. Yet, it is often employed by application programmers, compiler writers, and even hardware designers themselves. While the concept will be described generically in this section, the technique is used by at least two hardware vendors, Alliant and Floating Point Systems. In the case of the Alliant, this concept of "algorithm pipelining" is employed by the compiler. In the case of FPS, the concept is the basis of programming methodology used in programming in the assembler for the FPS line of array and attached processors.

Algorithm pipelining is the extensive use of the linking of functional units through software decomposition. Chaining, as described in the pipelining discussion, could be considered the very simplest form of algorithm pipelining. This was described as a hardware assisted process of streaming the output of one functional unit into another functional unit -- thereby linking the two already segmented units into a bigger functional unit performing two tasks instead of one. Software pipelining is simply the logical extension of this process through the use of both hardware and software to link input and output of functional units of any kind.

One form of algorithm pipelining can occur when a computer design allows for more than one instruction to be processed simultaneously. The HEP, for example, can provide a linking between processes causing a form of chaining of one instruction stream with another. The FPS-164/264 issues single instruction words which are coded to allow as many as 13 simultaneous instructions to be issued at the same time (cycle). Through clever use of main memory, data pads, and table memory, one can write an inner product loop on FPS machines that is one cycle in length. The essence of the loop is the chaining of functional units. Once the loop is full the various functional units are operating simultaneously, but on different data operands in a pipelined fashion. A third example of algorithm pipelining results from the innovative use of parallel CPUs. This form of algorithm pipeline is illustrated by an example from the Alliant Computer

Company's Product Summary [2-7]. The example deals with vector data dependency which is strictly ruled out when utilizing SIMD vector processors. Consider the following scalar loop, typical of large application programs.

```
         DO 24 I = 1, IEND
         X = FT(I)*FLOAT(N)
         X2 = 2. * X
         X21 = X2 - 1.
         DF = X21/X2
         DF1 = DF * R
         DF2 = DF1 * DF1
         FF = F(I) + DF2          <========
         F(I+1) = FF              <========
         AF = ABS(DF2/F(I+1))
         IF(AF.LE.EPS) GOTO 25
    24   CONTINUE
```

The vector F, which is manipulated in the loop, violates the fundamental principle of a vector operation. The "result" vector F is not independent of the "input" vector F. In particular, F(I+1) depends on F(I) as highlighted. Using multiple CPUs in MIMD (asynchronous) fashion, it is still possible to achieve a speed up of this inherently non-vectorizable loop. Imagine three CPUs are being employed, and that they begin to operate on copies of the loop for successive values for I. The first CPU will be able to process the entire first pass of the loop without interruption. The second CPU will have to pause at the statement with the arrow (above) because F(2) is not yet available. The pause (ignoring cache or memory conflicts) will be one cycle while the first CPU finishes the calculation of F(2). CPU number 3 also paused at the same statement awaiting the calculation of F(3) in CPU-2. In the mean time, CPU has begun the fourth loop iteration which requires F(4) from CPU-3. This will be calculated and available by the time it is required. This process repeats until all 3-CPUs are running simultaneously on different iterations of the loop. After the initial three loop iterations there is very little pause time. One can view this process as algorithm pipelining. Each pass through the loop enters the 3-CPU computation as CPUs are available and is process as data dependencies are resolved by the exchange of data between CPUs. This latter process is accomplished by the compiler through extensive pre-execution dependency analysis. The Alliant hardware facilitates the interchange of data and MIMD processing with little or no synchronization overhead beyond the wait time for data dependency resolution.

************* **IMPORTANT CONCEPTS** *************

o There is an ever increasing requirement for increased computer power:

  - more refined gridding or more dimensions
  - more complex and realistic physical models
  - improved user interfaces drive more complex processes
  - creation of optimal design tools from traditional analysis programs

o Traditional computer designs have hit fundamental technological barriers in chip technology requiring innovation in computer architecture in order to meet computational demand.

o Three common forms of parallelism in computer architecture have been used for a number of years.

  - simple replication of CPUs or functional units
  - simultaneous operation of functional units
  - pipelined processes

o The manipulation of vectors has been the key approach to today's supercomputers. The most common approach has been SIMD (single instruction, multiple data stream) processing. The advent of parallel CPUs allows for MIMD (multiple instruction, multiple data stream) computing.

o Novel approaches to processing instructions have led to unusual computer designs and processes. The HEP is an example of pipelined instruction streams that support MIMD processing.

o With the advent of multiple CPUs, multiple functional units, and MIMD processing, a new form of optimization, "algorithm pipelining" is possible on many hardware designs.

****

2.2 SUPERCOMPUTER ARCHITECTURE

2.2.1 Scope

The term supercomputer almost defies definition. Debates rage among vendors, users, and even procurement organizations, as to what is, and what is not, a supercomputer. The glossary in Appendix A offers the following definition:

> The class of general purpose computers that are both faster than their commercial competitors AND have sufficient central memory to store the problem sets for which they are designed.

The notions of "fast" or "computer power" are equally elusive. In the final analysis, a computer that can solve the most meaningful problems, that are otherwise computationally intractable, with reasonable real-time throughput, will be included in the class of supercomputers. Those machines that are considered "supercomputers" are often endowed with some technological and architectural features that set them apart from more commonly available machines. It is not unusual for machines sharing such distinguishing features to be included in the class of supercomputers. This can lead to some false or misleading claims. For example, today's supercomputers generally have pipelined architecture. Many of today's mid-range computers are using the same approach to performance improvement. Some call themselves small supercomputers -- not because of the above definition, but because they share "vector" technology with the larger machines. This is a misnomer that leads to some confusion.

As of this writing, there are 5 companies currently marketing commercially available supercomputers. They are Cray Research, Inc. (CRI), ETA Systems/CDC, Fujitsu/Amdahl, Hitachi, Nippon Electric Co. (NEC). Even though IBM tends to avoid the term supercomputer, one could justifiably include the 3090/600 with vector facility. Similarly, one could include the National Advanced Systems 9100 series, a compatible competitor of the 3090/VF. Once these machines have been included in the discussion, then other more recent computers might also be added, such as the UNISYS vector processor which can achieve higher peak ratings than the latter machines. In similar fashion, one could begin to include other vendor machines, and soon, the list is so long that one loses sight of the original goal: to overview the "very" top end mature supercomputers available. With this goal in mind, we have decided to concentrate on the following machines as examples of modern supercomputer technology:

CRAY-1, CRAY X-MP, CRAY-2
CYBER 205, ETA GF-10
FUJITSU VP 200/400 (AMDAHL 1200/1400)
HITACHI S-810
NEC SX-2
IBM 3090/VF

These machines are currently commercially available, general purpose, and have a market focus toward engineering/scientific computing.

In this section an overview of supercomputer architecture is given, but from a computational perspective. In Section 2.3, a discussion of parallel computers and minisupercomputers is given with the perspective of identifying likely trends in computer architectures in the near future. In the remainder of Section 2.2, those elements of computer design that directly impact scientific computation will be emphasized. An expert in computer hardware might find the descriptions somewhat lacking in detail, yet detail is expressly being avoided, except as it impacts computation and algorithm design.

2.2.2 Growth in Computational Power

In Section 2.1.1 the need for increased computational power was discussed. This requirement for more power pervades many (if not all technologies.) The economic realities of the market at this high end of computing has changed dramatically. One of the most successful computer companies today by any financial measure you pick, has to be CRI. Nevertheless, this is a high risk-low volume market, and the number of players has been very small. The birth of the industry must be credited to the early financial support of the government labs, who saw a direct relation between national security and the ability to "compute" scientific phenomena. Indeed, early models of US supercomputers invariably go to the government labs. In Europe, the government influence seems to be exercised more frequently through academic institutions. A brief examination of the computational power afforded by modern supercomputers reveals an astonishing growth in floating point computation rates and memory size. Figure 2-10 illustrates this growth in terms of a fundamental metric often used (and sometimes abused) by supercomputer manufacturers. This metric is a measure of the "peak" possible performance in floating point operations per second. This parameter is often given in millions (MFLOPs) or, more recently, billions (GFLOPs) of floating point operations per second.

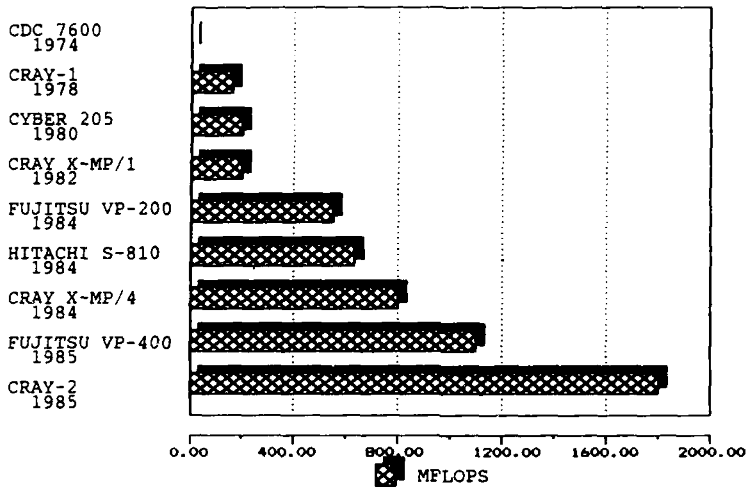


Figure 2-10
Growth in Power: *MFLOPS*

There are two facts worthy of note. First, the increases in computational growth, of say, a factor of 10, are occurring with shorter and shorter time constants. That is, what took the industry 6 years to accomplish in improving performance, it now is achieving in 3 years. Second, the performance improvements are not being achieved by electronic circuitry alone. For example, the CDC 7600 could achieve a 2-5 MFLOP computational rate with a machine cycle time of 27.5 nanoseconds. The CRAY-1 could achieve 160 MFLOPs with a cycle time of 12.5 nanoseconds. Thus, a factor of 32 improvement in peak performance with only a factor of 2.2 improvement in machine cycle time. The trend, as you go down the chart, is *similar*. *Both of these facts* are attributable to the same source, the computer architect. Once the market place accepted the burden of computational complexity resulting from architectural changes, computer manufacturers could easily increase "peak" performance potential. This "burden" of complexity will be the focus of Chapter 3.

To match the ever increasing capability of processing floating point operations, vendors have had to supply improved memory capacity. Figure 2-11 reveals that this growth in capacity has been quite impressive as well.

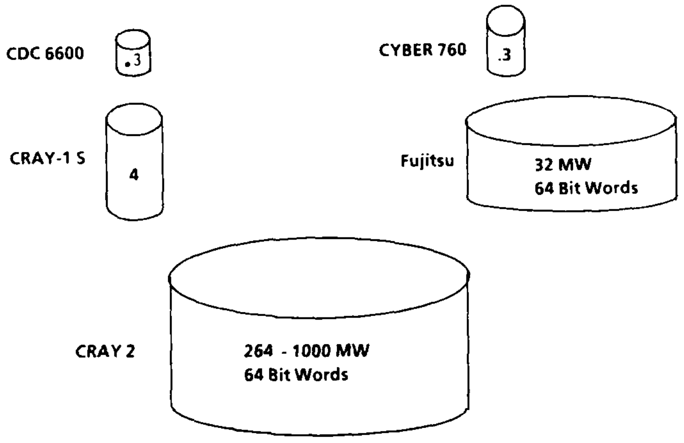


Figure 2-11
Growth in Memory

How are these phenomenal improvements accomplished? What are the specific design tricks being employed by architects to achieve these performance increases? In reality the approaches are both similar and yet very different among current supercomputers. Like a baker making a batch of cookies, when he is done, no one questions that the result are cookies; yet, the ingredients might be very different between chefs. In the next section, the salient features (ingredients) of a supercomputer CPU will be described. In addition, the various supercomputers will be contrasted and compared according to their respective implementation of each feature. In Chapter 3, the impact of each feature will be examined on various basic computational kernels often used in scientific computation.

2.2.3 An Overview of Supercomputers' CPUs

In Figure 2-12 a simple generic diagram of a supercomputer central processing unit is displayed. While several vendors of supercomputers offer multiple CPU machines, the discussion initially will concentrate on a single CPU. Multiple CPU machines from CRI and ETA Systems will be discussed in Section 2.2.4. The figure itself is more symbolic than an actual replica of a manufacturer's hardware diagram.

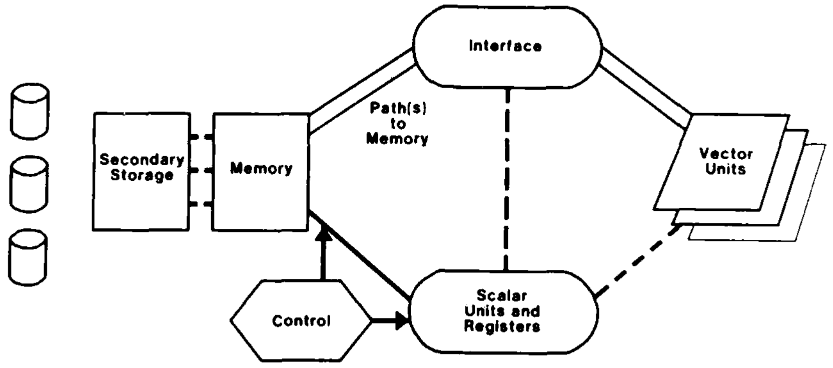


Figure 2-12
A Supercomputer CPU

The following features will be discussed at length:

SCALAR AND VECTOR FLOATING POINT UNITS
PRIMARY AND SECONDARY MEMORY
INTERFACE (BETWEEN MEMORY AND VECTOR UNITS)
PATHS TO/FROM MEMORY
CONTROL (INSTRUCTION) PROCESSOR
SECONDARY STORAGE.

In, and of itself, no feature above is a sufficiently important parameter for assessing computer performance. The interplay or "balance" between these elements, however, give the computational character of a given computer. The most distinctive feature of modern supercomputers is their orientation toward processing "vectors" or arrays of elements as operands. For years the computational bottleneck in scientific computing was the processing of floating point computations. The CDC 6600, for example, tried to improve this bottleneck by using two floating point multiply units. Later the CDC 7600 exploited the pipelined concept in the functional units. Interestingly enough, the floating point units on the 7600 were, all too often, left idle. The bottleneck to computation was the rate of instruction processing associated with the overhead of fetching and storing each pair of operands and/or result. The first modern supercomputers (the CDC Star-100 and the CRAY-1) circumvented this "instruction-issue" bottleneck by extending the instruction set to include vector operations. This coupled with a high bandwidth connection between the vector units and memory, through an interface (buffer or registers), characterizes modern supercomputer CPUs. In these, and subsequent supercomputer designs, one single instruction could launch a process that operated, not on one, but many operand pairs. The production rate of floating point operations became a much more meaningful measure of performance than did machine instruction rates. With the advent of the "vector" computer, the performance rating of millions of machine instructions per second (MIPs) gave way to the MFLOP mentioned earlier.

Each of the characteristic features of modern supercomputers deserves further discussion. To that end, a brief discussion of each feature is offered below. Many of the principles discussed will be utilized in the algorithm discussions in following chapters.

The Scalar and Vector Units

The most successful supercomputer designs have struck a very delicate balance between the scalar processing speed and the vector processing speed. All of the supercomputers discussed exploit pipelining in their vector floating point units. In the case of the CRAY series, scalar floating point computations also use the floating point vector units. In other designs separate scalar floating point units are included. In many ways the marriage between scalar and vector processing is more philosophical than one would imagine. Some designs were forged out of a priority to create a "vector" computer, with scalar computing provided as a necessary, but unimportant, adjunct, much like the letter "q" on an English typewriter. It is necessary, but used so little that it can be placed in a remote area of the keyboard. The CDC Star-100 is a prime example. Its original design was, in philosophy, an attempt to implement the APL computer language in hardware. This language was designed to program and manipulate vector/matrix related computations. Scalar instructions in this environment were viewed as unnecessary and to be avoided. Early usage was primarily in Fortran, and revealed that scalar performance was a far too critical ingredient in program efficiency to be ignored. In an effort to improve the scalar/vector balance, the CYBER 203 was created. The 203, however, was quickly followed by the CYBER 205 with improved vector speed over the STAR-100. The result was that the 205 had slightly poorer balance between scalar and vector rates than the 203, but still much better balance than the Star-100. The CYBER 205 has 32 times the scalar speed of the Star-100 and only 10 times the vector speed. Even with this improvement in scalar speed the CYBER 205 is often criticized as being too slow in its scalar processing for many scientific applications.

Another philosophical approach to supercomputer design is to take a good scalar computer and "add" to it the salient features of vector processing (vector instructions and pipelined vector units). This approach has been adopted by IBM and its Japanese competitors. The philosophy can be observed in some minisupercomputer designs to be discussed later. The efficacy of such an approach varies with its implementation. Yet, it is often observed, that machines with poorly designed vector implementations, often suffer from some data movement bottleneck typical of poor internal bandwidth which , in turn, is related to the scalar design. That is, when a vector processing capability is superimposed over a fundamentally scalar design, the bandwidth of internal data flow (from memory to and from the functional units), inherent in the original scalar design, cannot support the vector data stream requirements of the vector floating point units.

Probably one of the most significant features of the CRAY-1 is the balance of vector and scalar processing. Not only is the scalar speed fairly fast relative to the vector speeds, but the internal bandwidth of data flow to support vector instruction execution, was far superior to its competitors at the time of its introduction. All successful vector oriented processors have achieved "good" scalar/vector balance. This balance does vary, and it is worthwhile to look carefully at the implementations of vector processing found in several machines. To begin, the vector units will be examined more closely.

The number of floating point units and their basic result rates differ among the various vendors as indicated by the following table. The number of segments in the pipelines also effects the startup time. Rather than list the number of segments another parameter will be discussed later that can be more useful in judging the effects of variations in segmentation among various machine architectures. Another feature, that is critical, is the "chaining" or "linking" of results from one floating point unit into another. (refer to Section 2.1.3, Figure 2-7.)

Vector Floating Point Characteristics

COMPUTER	VECT. UNITS PER CPU (+,*, or /)	PIPE CYCLE* in nanosec.	CHAINING
CRAY-1	2	12.5	yes
CRAY X-MP	2	8.5 (9.5)	yes
CRAY-2	2	4.1	no
CYBER 205	1, (2, 4)	20.0	(yes)
ETA-10/E	2	10.5	yes
FUJ. VP 100*	3	14.8 (15.0)	yes
FUJ. VP 200	3 (6)	7.5	yes
FUJ. VP 400	3 (12)	7.5	yes
HIT. S-810/20	2	14.0	yes
IBM 3090/VF	2	17.2 (18.5)	yes
NEC SX-1	2 (8)	7	yes
NEC SX-2	2 (16)	6	yes

*parenthetical cycle times indicate the original times when the machine was introduced. The current upgraded cycle times are listed whenever possible. In the case of Fujitsu and CYBER 205, the pipe cycle is not the "major" cycle, or machine instruction issue cycle, but the peak rate at which the floating point pipes can produce results.

TABLE 2-4

Unfortunately, one cannot infer peak performance directly from the table above, as one might expect. For example, the Fujitsu VP-200 is listed with 3 functional unit pipes. The three indicates add, multiply, and divide. Each add pipe is actually two pipelines in parallel running at a 7.5 minor cycle time. The computer's major cycle time (i.e. instruction issue cycle time) is 15 nanoseconds. Based on this one might conclude that an asymptotic performance rate of 800 MFLOPs is attainable on the VP-200 (i.e. 6 pipes with 7.5 nanosecond minor cycle). In fact, only two of the three pipelines are allowed to operate simultaneously, thus the real peak performance rate is 533 MFLOPs. Taking into account these anomalies, and each machine has its own set of anomalies, the following table lists the peak performance rate in MFLOPs for each computer in Table 2-4. This should be interpreted as the rate at which each computer is guaranteed not to exceed, and not as a meaningful measure of performance.

Peak Performance Rates

COMPUTER	SINGLE CPU PEAK MFLOP RATING
CRAY-1	160
CRAY X-MP	233 (210)
CRAY-2	488
CYBER 205	200 (2-PIPE, 64-BIT)
ETA-10 (1986)	350
ETA-10/E	415
ETA-10/G (1988)	625
FUJ. VP 100	271 (267)
FUJ. VP 200	533
FUJ. VP 400	1067
HIT. S-810/20	630
IBM 3090/VF	116 (108)
NEC SX-1	570
NEC SX-2	1300

*Note: The parenthetical entries indicate performance figures for the original performance before recent improvements in cycle times.

TABLE 2-5

To achieve peak performance for an entire machine, the single CPU MFLOP rating is usually multiplied by the number of CPUs. With this approach one could project CRAY-3 and ETA Systems performances into the 5 to 15 GFLOP range. To appreciate the rapid change in this environment, at the time of this writing announcements are pending on a 1.76 GFLOP single CPU capability from Fujitsu, 2 GFLOP/CPU (4-headed) Hitachi, and multiple headed NEC machines. It is also considered likely that IBM will enter this market with a "true" supercomputer as opposed to a retrofitted 3090. It is very likely that by early 1990 no machine under 10 GFLOPS will be considered a supercomputer. A conservative estimate would be that by 1995 there will be TeraFlop (1,000,000 MFLOP) general purpose supercomputers.

So far, the data presented lacks a dimension of performance related to the degradation that can occur when one does not use long vectors on vector computers. In reference [2-8], the term "depth of parallelism" is defined. When using an N-segment pipelined functional unit, an N-fold concurrency is being employed. If, in addition, a number, M, (of N-segmented) pipes is being used, the "concurrency" is N*M. That is, there is the possibility of N*M operand pairs being in some stage of computation in a given cycle. Intuitively, one suspects the higher the depth of parallelism the greater the penalty for not providing long vectors to operate on. A more elegant and more easily verified measure of this degradation is given by Hockney and Jesshope [2-4]. The term is called $N_{1/2}$ (N sub a half). Given an operation (or even an algorithm), $N_{1/2}$ is defined to be the length of a vector to achieve one half the asymptotic peak performance for the given operation (or algorithm). One can see that this measure is a function of pipe segmentation and chaining for a given operation or set of operations. Perhaps the best way to visualize this term is graphically. Figure 2-13 displays the performance in MFLOPs for the vector triad , a*X+Y (a scalar, "a", times a vector "X" plus a vector "Y", or called SAXPY after reference [2-10]), as a function of vector length. The curve is obtained by modeling the time of a vector operation as follows:

$$T = S + K*N, \qquad (2\text{-}1)$$

where S is the startup time to fill the pipeline, and K is a constant (related to the pipe cycle time). Solving this equation for computation per unit of time yields,

$$N/T = 1/(S/N+K).$$

As N approaches infinity, N/T approaches the asymptotic rate, R, for the vector process, as indicated in Figure 2-13. In Section 3.2 a curious relationship between the depth of parallelism and $N_{1/2}$ is derived for pipeline operations.

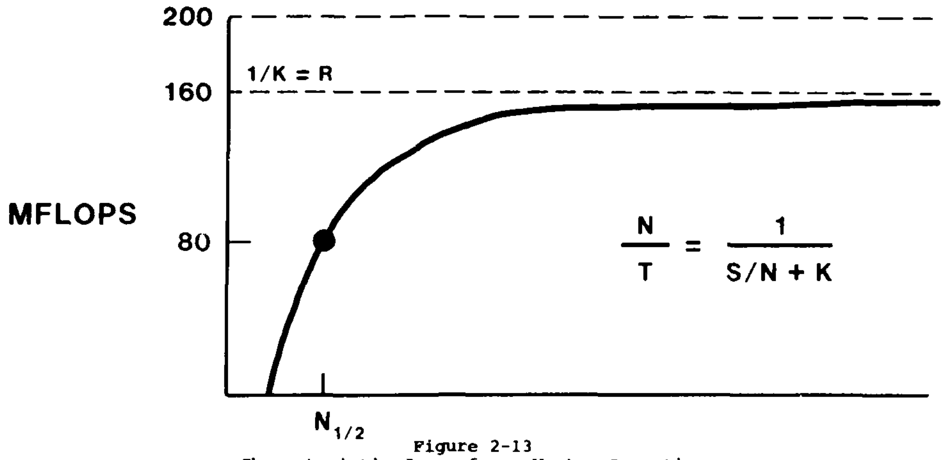


Figure 2-13
Characteristic Curve for a Vector Operation

Figure 2-13 is an idealized curve for most vector computers because the timing model given by Equation 2-1 represents an idealized linear model. There are many reasons why the true time for a vector operation is not necessarily linear. On some machines, such as the IBM 3090/VF, the data for the vector operation process is fed by a cache (see glossary). This can cause deviation from the linear model due to cache "misses". (When feeding data to functional units or registers from a cache, the required data is sometimes not available in the cache. When this occurs, an automatic process is begun by the system to refill the cache with a section of memory containing the wanted data. This causes considerable delay, and is often termed a cache "miss", referring to the interruption caused by the cache refill algorithm.) Another more common deviation from the linear model comes from the use of registers (to be described below in greater detail.) For example, on the CRAY series of computers, vectors are handled in blocks of 64 words, corresponding to the depth of the vector registers. Thus, if a vector of length 250 were to be processed, it would be done by dividing it into three vectors of length 64 and one vector of length 58. The precise timing model for a CRAY-1 is given below after Bucher [2-9].

$$T = S_{OUT} + N\,(S_{IN}/64 + T_{EL}), \qquad (2\text{-}2)$$

where S_{OUT} is the initial startup time for the whole process (the outer loop), S_{IN} is the startup time of the inner loop of length 64 or less that cannot be overlapped with other vector instructions, and T_{EL} is the time to process one result element. Clearly, these parameters are both hardware and software (compiler) related. The plot of Equation (2-2) is given in Figure (2-14). The parameters of the model vary with the vector operations being formed.

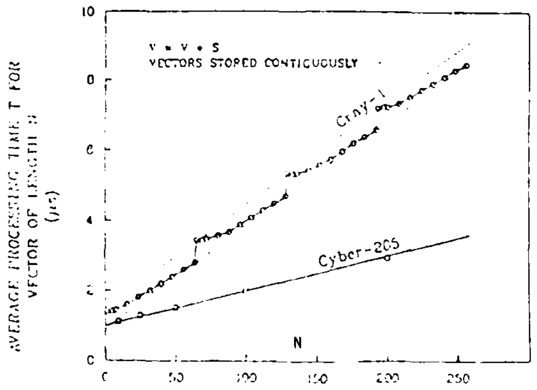


Figure 2-14
An Actual Timing Model: CRAY-1 (CFT 1.09)

The characteristic curve in Figure 2-13 would look about the same except it would have small jump discontinuities for N equal to a multiple of 64. For most computers and applications, a linear timing model will be adequate in describing performance. It is interesting to note that the $N_{1/2}$ is roughly the same under either model for a given operation and machine. In Chapter 3, both the asymptotic computational rate and the $N_{1/2}$ will be used to examine performance. To give some idea of how these parameters can be used, consider the SAXPY operation. Two earlier supercomputers competed in the marketplace for years, the CRAY-1 and the CYBER 205. They have very different characteristic curves, as displayed in Figure 2-15.

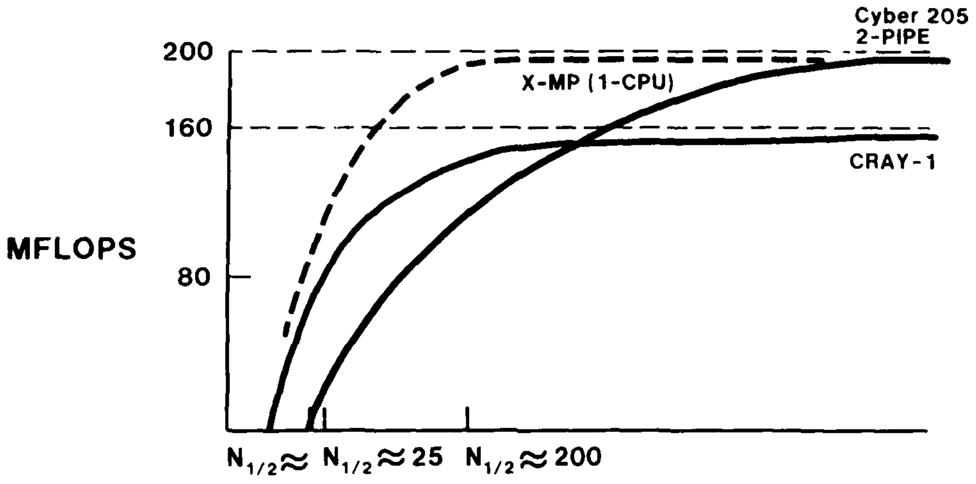


Figure 2-15
CRAY-1 and CYBER 205

The CYBER 205, 2-pipe using 64 bit arithmetic, has the higher asymptotic rate (200 MFLOPS vs. 155 MFLOPS). However, the $N_{1/2}$ for the CRAY-1 is on the order of N = 25, while it is over 200 for the CYBER 205. This means that if we were to perform the SAXPY operation on vector lengths of 200 the CYBER 205 would run at 100 MFLOPS while the CRAY-1 would be running very close to its maximum. In fact, a rule of thumb would be if the vector length is over 3 times the $N_{1/2}$, the resulting vector operation will be very close to its maximum. If on the other hand, the average vector length is less than $N_{1/2}$, the resulting operations would be only using less than half the potential power of the machine. During the time the CYBER 205 and the CRAY-1 competed in the market, most scientists had not worried about vector length. By the time scientific programs began producing longer vectors, the CRAY X-MP was released with a higher asymptotic maximum (200 MFLOPS) with little degradation in $N_{1/2}$.

The relationship between $N_{1/2}$ and the peak asymptotic performance rate is a very curious one. At face value, one would desire $N_{1/2}$ to be small, and the asymptotic rate to be high. However, in a purely scalar environment, $N_{1/2}$ is one. By the introduction of pipelining and/or parallelism the architect is able to increase maximum performance with the same hardware circuit technology, but at the expense of scalar degradation due to increased $N_{1/2}$. Each computer design is the result of alternate strategies of parallelism, chip technology, and internal bandwidth. Table 2.5 below lists the $N_{1/2}$ and the asymptotic rate of various computers in performing a SAXPY operation. The results are due to Dongarra [2-11], and are the results of Fortran testing. Considerable and significant differences can be obtained by carefully coding in assembler. It is also possible to improve the results with more mature Fortran compilers (e.g., the CRAY-2 result below casts dubious concerns on the maturity of the compiler used.) A large $N_{1/2}$ is only justified by a significantly large performance potential. Obviously, if the value of $N_{1/2}$ is so large that no reasonable program can produce vector lengths large enough to take advantage of the potential peak performance, the high MFLOP rating is meaningless. These issues will be explored in greater detail in Chapter 3.

Values for $N_{1/2}$, (Fortran SAXPY)

COMPUTER	$N_{1/2}$	ACTUAL PEAK PERFORMANCE
CRAY-1	20	45
CRAY X-MP	37	101
CRAY-2	30	55
CYBER 205	238	170
FUJ. VP 100	200	140
FUJ. VP 200	120	190
IBM 3090/VF	34	53
NEC SX-1	30	240
NEC SX-2	80	575

TABLE 2-6

************* **IMPORTANT CONCEPTS** *************
Scalar and Vector Units

- The balance between scalar speed and vector speed has proven to be an important characteristic of modern supercomputers. The greater the disparity of performance between the scalar units and the vector units, the worse the "balance".
- Peak performance ratings can be very misleading. The number of pipelined units required to achieve peak performance, the startup time (and hence $N_{1/2}$), the cycle time, and the use of cache interfaces are but a sampling of the hardware features that can impact actual performance regardless of peak ratings.
- Managing the instruction issue to achieve chaining of pipelined units can be critical to performance in many hardware designs.

****

Primary and Secondary Memory

The improvements in chip and circuit technology have allowed manufacturers to develop large capacity memories at very low access times. The large and growing disparity between memory speeds and secondary storage speed (e.g. disk) can create a very pronounced degradation in throughput in programs with large amounts of data. Quite often in scientific computation the storage requirements of intermediate data, generated through the course of the computation, can exceed the storage requirements of the original or final data. In such a situation tremendous performance gains can be achieved by providing a secondary or tiered memory structure. The trend to tiered memory seems to be a necessary approach to deal with the shortcomings of traditional disks. The idea behind tiered memory is both economic and technological. Secondary memory, if properly used does not have to be accessed often. It can be made larger and cheaper with slower access times than main memory, but with much higher bandwidth than disk. From a user perspective, the lack of sufficient memory, the availability of secondary storage, or the use of massively large memory systems present program design challenges. This is particularly a problem when migrating software from an "old" technology to a "newer" one. For example, early users of CRAY-1's often wrote to disk when more memory was available, and early users of the CRAY-2, the largest main memory machine available, would convert programs by having disk reads and writes changed to memory reads and writes. This gave an instant improvement in performance, but is not necessarily the best use of large memory machines.

The size of memories in today's supercomputers are really a function of chip technology. Those companies that are vertically integrated such as the Japanese manufacturers and IBM, have an advantage in accessing their latest technologies sooner than the competition. CRI has been able to package denser memories through advanced cooling techniques as opposed to advanced chip technology. The table below gives the relative size (in terms of 64-bit words) of main memories for the computers under discussion as of January 1987. In addition, two other parameters are provided: the time to access a single word, and the single vector bandwidth. This latter rate, given in words per second, is the asymptotic rate for accessing a vector of contiguously stored components. One should take note that some of the computers listed allow for more than one vector fetch/store simultaneously. This is not accounted for in the bandwidth data, but addressed in discussion on "paths-to-memory" (to follow). Most main memories have a latency time -- a length of time required between successive data fetches. To ameliorate the effects of this latency time, memory is often grouped into "banks" or logical partitions. In this manner vector data can be accessed continuously (i.e.

without interruption) provided the data does not come from the same bank or very recently accessed bank. This can be a problem for accessing vectors with power of 2 strides (for often the number of banks is a power of two.) The number of banks for each computer is listed in the table as well.

Main Memory Sizes

COMPUTER	size MW	no. of banks	bank wait (nsec)
CRAY X-MP	16	64	76.5
CRAY-2	256	128	140-170
ETA-10/E	128	n/a	214
FUJ. VP 200	32	128	55
FUJ. VP 400	32	256	55
HIT, S-820	32	128	70
NEC SX-2	32	512	40

TABLE 2-7

Memory plays a very crucial role in applications such as computational fluid dynamics. However, large memory without a balance of computational power could create a step backward. The use of large memory machines and the technique of employing tiered memory machines are active research topics today. Memory size has long been a limiting factor in computational fluid dynamics programs as will be observed in Chapters 4 and 5. In a subsequent subsection we will examine secondary storage and its relation to main memory in more detail.

Interface (between Memory and the Vector Units)

The flow of data from the memory to the computational units is the most critical part of the computer design. The object is to keep the functional units running at their peak capacity. Through the use of proper interface (combinations of vector registers, caches, local memories, or buffers) the maximum throughput can theoretically be attained. In many ways the various interfaces are simply additional types of memory, with high bandwidth and multi-ported connections to the vector units.

All the machines we have considered, thus far, utilize some form of vector registers except the CYBER 205 and the ETA GF-10. These latter machines employ hardware directed "buffers" which provide a steady flow of data minimizing interruptions from memory. Among the supercomputers under discussion, these two designs are the only ones that can be classified as memory-to-memory computers. That is the functional vector units operate on memory-stored data directly. In the case of the Cyber 205 and the GF-10, the vector data is defined by its first-word location in memory and vector length.

The most common interfaces are "vector registers," first employed and patented by Seymour Cray for the CRAY-1. Registers have been used in scalar computer designs for years. The need to store temporary data in accessible fast memory has resulted in registers employed in most conventional computer designs. Since modern supercomputers manipulate "vectors" the idea of registers for easily accessed packets of data is most reasonable. In many machines the functional units operate only on operands stored in registers, and this is a common technique in the various register oriented vector computers. The CRAY-1 employs 8 vector registers of length 64 words. This results in long vectors being manipulated in sub-blocks of 64 words. Table 2-8 lists the buffer or register characteristics of supercomputers being considered.

Register and Buffer Characteristics

COMPUTER	SINGLE CPU REGISTER CONFIGURATION (64-bit words)
CRAY X-MP	8 X 64 words
CRAY-2	8 X 64 words, plus 16K local memory
CYBER 205	buffer
ETA-10	buffer
FUJ. VP 200	reconfigurable 8 X 1024 . . . 256 X 16
FUJ. VP 400	reconfigurable 16 X 1024 . . . 512 X 16
HIT, S-820	32 X 256 words
IBM 3090/VF	8 X 64 words
NEC SX-2	40 X 256, 32 reconfigurable e.g. 64 X 128

TABLE 2-8

In some minisupercomputers and, in the case of the IBM 3090/VF, vector registers are fed from a high speed (8K 64-bit words) cache. (See Figure 2-16.) In the case of the IBM 3090/VF, this cache has 8K words (64K bytes).

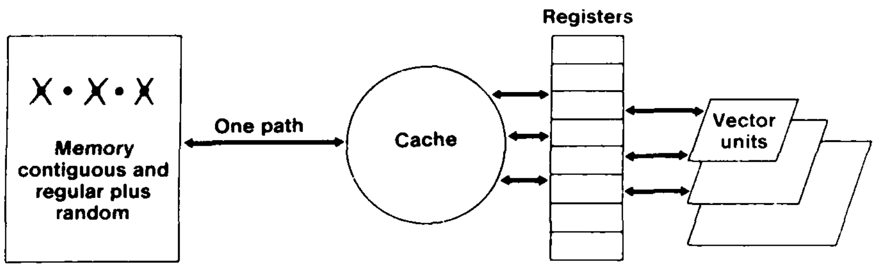


Figure 2-16
Cache-Interfaced Registers

The term cache refers to another form of storage. The term cache also infers that this type of storage is automatically filled and emptied according to a fixed scheme defined by the operating system and/or hardware design. For example, if a certain area of memory is to be accessed in a cache based system, either it is already in the cache or it is not. If the latter is the case, then there usually is a fixed algorithm for how much of cache data is to be displaced by main memory data in a predescribed neighborhood of the requested data. This should be distinguished from, say the CRAY-2, which provides a 16K "local memory" in addition to registers. (See Figure 2-17.)

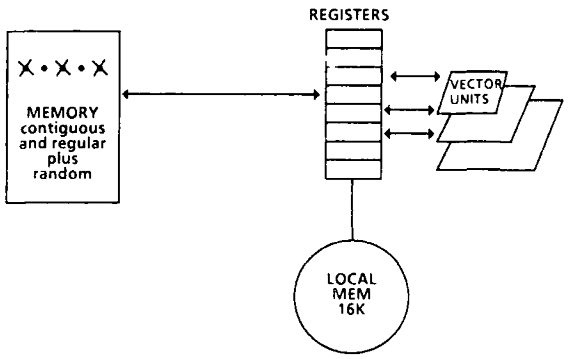


Figure 2-17
Local Memory Enhanced Registers

The local memory on the CRAY-2 is written into, and fetched from, through user control. Of course, compiler developers at CRI are also using the local memory to enhance performance of executed programs to the benefit of Fortran users. The local memory is not a required interface between the memory and the registers, but rather an auxiliary storage place. Thus, it would be incorrect to call this local memory a cache. In attempting to write efficient algorithms in a cache-based system, users often learn the eccentricities of the cache algorithms and develop appropriate algorithm strategies to prevent frequent filling of cache memory (i.e. algorithms that avoid cache misses.) With either cache or with local memory, both the complexity and the flexibility of the system is increased. The greatest advantage of such features occur when mature compilers deal effectively with these new hardware features. Unfortunately, mature compilers are not common place on new machines with innovative architectures. It will be observed later that to relegate the use of special architectural features solely to the domain of languages and compilers could quite easily relegate performance to less than "super."

Paths-to-Memory

Referring to Figure 2-9, once again, one observes that the connection between memory and the interface is termed "paths-to-memory." This is not necessarily a standard term of computer jargon, but it conveys the sense of data access that algorithm design must consider. All computer algorithms ultimately come down to series of operations. The most efficient use of the current generation requires a high percentage of the operations being executed in "vector mode" (i.e., on vector operands). Most elementary operations involve two operands and one result. If the operation is a vector operation, then quite often the operands are vectors and the result is a vector. In order to keep the functional units busy, a well designed machine would have a high degree of memory access afforded by extremely high bandwidth path-to-memory. In fact, more than one path-to-memory is desirable. The Table 2-9 gives a listing of the number of vector paths (i.e. read/store pipes) from the vector interface to main memory.

Paths-to-Memory

COMPUTER	NO. OF PATHS-TO-MEMORY	# PATHS/# PIPES	LATENCY in cycles
CRAY-1	1	.5	11
CRAY X-MP	3	1.5	14
CRAY-2	1	.5	35-50
CYBER 205	3	1.5	50
ETA-10	3	1.5	n/a
FUJ. VP 200	1 or 2*	.5 or 1	31-33
FUJ. VP 400	1	.5	31-33
HIT, S-820	8	1.0	n/a
IBM 3090/VF	--	associative	--
NEC SX-2	12	.75	n/a

*The Fuj. VP-200 has two paths to-memory for contiguously stored vector elements. For vectors with a stride or random storage only one path is available.

TABLE 2-9

In all cases, the process of memory access is a pipelined process. The second column in the Table 2-9 is provided to normalize the data. Several of the machines listed have a large number of paths due to the fact that they employ multiple pipes to achieve high performance. Such machines require multiple hardware paths to fetch one logical vector operand. For example, if a hardware design were to employ two tandem multiply pipelines to achieve its peak performance, then to fetch two vector operands, four paths would be required to achieve peak speed. Thus, in column two of the table, the number of hardware paths-to-memory is divided by the number of floating point pipelines being served. The third column in the table gives the latency of the pipe; that is, the number of machine cycles required to receive the first element from memory during a vector fetch.

The number of paths-to-memory has a direct bearing on efficiency and algorithm performance. This can be illustrate by the following example. Again consider the SAXPY operation,

```
      DO 10  I = 1, N
10    Y(I) = A * X(I) + Y(I).
```

This fundamental loop, employed in many linear algebraic operations, displays unusually simple complexity. It is operating on two vectors and replacing one with the result. This requires two vector fetches and one vector store, provided N is smaller than the register size on a register oriented machine. If N is larger than the register lengths, the process would proceed as described in Equation 2-2. Assume that the computer to be used to perform this loop is register oriented, and that the multiply unit can be linked to the add unit. In the following sequence of figures all other features are held fixed, and the number of paths-to-memory is varied.

Each parallelogram in the figures indicate a vector operation. The x-axis indicates increasing time, measured in machine cycles. For simplicity we assume all operations (i.e. fetches, stores, and arithmetic) are pipelined with a fixed pipe length. The left most edge of a parallelogram indicates the pipe length by signifying the point in time at which the operation begins (top left corner) to the time the first element is out of the pipe (bottom left corner). The upper edge, and lower edge are of length N indicating the number of cycles required for the N elements of the vector to be read from memory (top edge) and received into registers (bottom edge). The last element of the vector enters the pipe (indicated by the upper right corner) and exits the pipe (as indicated by the lower right corner).

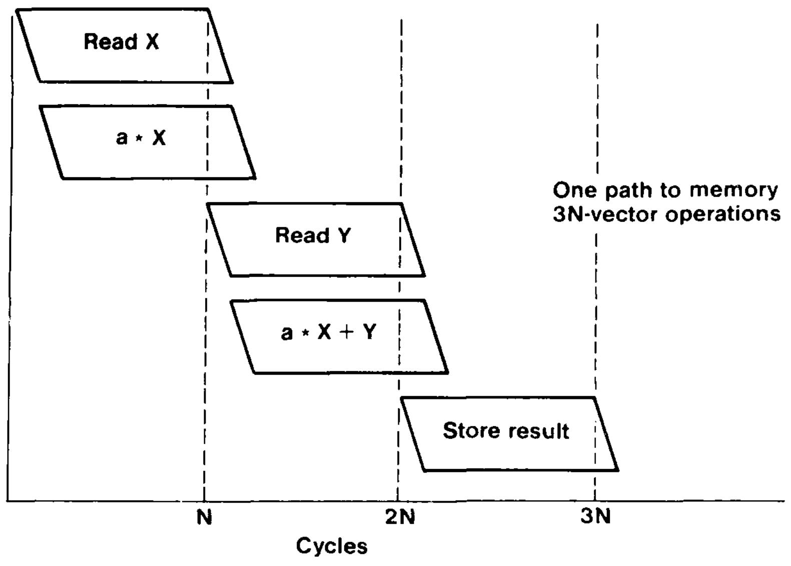


Figure 2-18
One-Path-to-Memory SAXPY

The figure displays the order of operation best suited to compute the SAXPY. It allows chaining the multiply result into the adder saving needless memory references by having to store the intermediate result. The entire operation takes order 3N cycles, neglecting startup times. One can observe, however, that each functional unit is idle two thirds of the elapsed time. The multiply unit, for example, is only used in the first order N cycles, and the adder is used in the second order N cycles only. The pacing element of the loop is memory referencing (reads and stores). Figure 2-19 displays the improvement gained by providing one more path. Now vectors X and Y can be fetch simultaneously; the multiply can be begun followed by chaining into the add. The only "dead" time occurs in the second order N cycles due to the store. By providing another path to memory (for a total of three), one can also chain the result of the add into the store and achieve an order N process. Thus, without any increase in speed in the functional units, a factor of three improvement in loop performance is achieved. If one reflects on the method of rating computer performance in peak MFLOPs, one can observe that a 1-path machine will have the same peak performance as quoted by the manufacturer as a 3-path machine with the same chip technology and functional unit structure. In fact, the performance is quite different by any measure when applied to a simple, yet fundamental, loop such as SAXPY.

Observe that the two-path loop looks very inefficient. For example, in the first N cycles there are 2 reads, a multiply, and an add running simultaneously, while in the second N cycles only the store is executing.

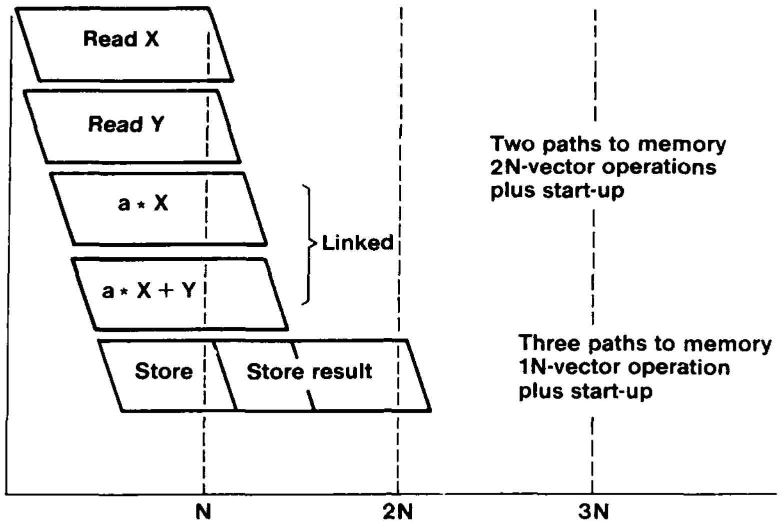


Figure 2-19
Two/Three-Paths-to Memory

Figure 2-20 illustrates that ...e can improve the asymptotic performance of this loop on a two-path machine by 50%. This is accomplished by dividing the vector into two sub-vectors of length N/2. The only requirement, for this to be successful, is that 1) the vectors be long enough to ameliorate the extra startup times, and 2) that there be sufficient vector registers. This same trick can be used to improve performance of a M memory referenced loop on an M-1 path machine. These types of optimizations can be critical to performance in key computation and are not often employed by compilers.

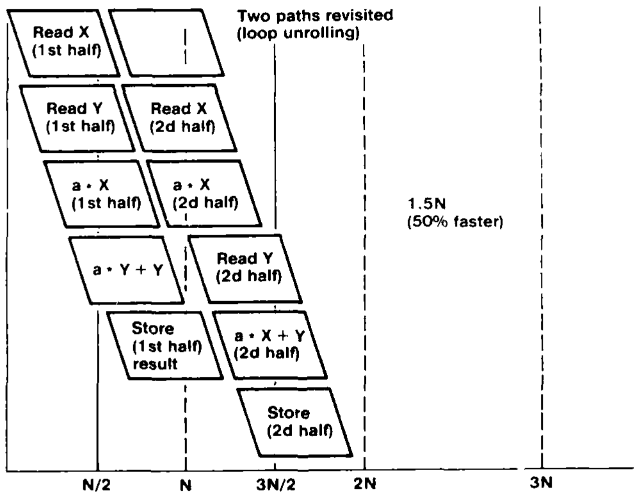


FIGURE 2-20
Two-Paths and a Trick

In Chapter 3, the impact of these types of performance variations will be assessed on several basic algorithms. Quite often judicious use of registers and an awareness of architectural features such as the number of paths-to-memory can lead to superior algorithm design and performance.

************* **IMPORTANT CONCEPTS** *************

Memory Interfaces and Paths-to-Memory

- The flow of data in a supercomputer CPU from main memory to the floating point and other functional units can vary. Currently, the following hardware options can be found:

  - buffers
  - cache
  - local memory
  - registers
  - combinations of the above

- An awareness of the above features in a particular hardware design can be extremely useful when designing algorithms and optimizing existing or selecting key computational kernels.

- Registers and local memories, through user control, can be very useful in storing vector temporaries and thereby minimizing memory references. Caches improve data flow in ideal situations, but cause user concern over algorithms that can regularly cause cache "misses."

- The number of paths-to-memory, if insufficient, can cause severe performance degradation if ignored. For complex loops, requiring many vectors and temporaries, the combination of a sufficient number of paths, judicious use of registers, and tricks like loop unrolling can be combined to make a significant improvement performance.

****

Control or Instruction Processors

Thus far, only the aspects of computer architecture that directly effect computation have been presented. In so doing, a number of hardware oriented technicalities have been avoided. Almost all aspects of the computer's systems control and instruction processing are important to some degree in computation and algorithm design. To attempt an in-depth study of all the instructions offered on 5 or 10 computers would be far beyond the scope of this treatise. The complexity of instruction processing alone would be a formidable topic. For example, the Fujitsu VP/Amdahl 1200, a computer that has scalar IBM compatibility, as well as vector processing capability, has 195 scalar instructions and 83 vector instructions. On the other hand the CRAY-1 has only half as many. Moreover, a machine line like the CRAY X-MP has three separate instruction sets for three models of X-MP; that is, the instruction set has been altered in successive models! In this section, the focus will be on the relation between the instruction processing and vector processing. The specific areas of interest are the following:

ALLOWABLE VECTOR DATA

SCALAR/VECTOR OVERLAP

UNUSUAL INSTRUCTIONS

(Readers interested in complete lists of instructions are referred to vendor manuals such as hardware reference manuals or technical overview manuals.)

To a mathematician or scientist, a vector is simply an ordered array of numbers. For a computer this array doesn't exist until it is designated by the memory location of its ordered components. Various supercomputers have limitations as to the flexibility allowed for the specification of vectors. There are three common complexities found in manipulating vectors stored in computer memories, as illustrated by Figure 2-21. The figure displays the storage of a (dense array) vector $A = (A_1, A_2, \ldots, A_n)$ in three different memory configurations. The `X' denotes that that storage location is not used by the vector data.

```
-----------------------------------------------------
: A  : A  : A  : A  : A  : A  : A  : A  : A  : . . . :
:  1 :  2 :  3 :  4 :  5 :  6 :  7 :  8 :  9 :       :
-----------------------------------------------------
```

Contiguously Stored

```
-----------------------------------------------------
: A  : X  : A  : X  : A  : X  : A  : X  : A  : . . . :
:  1 :    :  2 :    :  3 :    :  4 :    :  5 :       :
-----------------------------------------------------
```

Regularly Stored (Strided Storage)
(Stride of 2)

```
-----------------------------------------------------
: A  : A  : X  : X  : X  : A  : X  : A  : A  : . . . :
:  1 :  2 :    :    :    :  3 :    :  4 :  5 :       :
-----------------------------------------------------
```

Random

Figure 2-21
Types of Vector Storage

The retrievability of vectors stored by any of the three schemes is a function of hardware design. Contiguously stored data is the simplest and most natural of the three storage schemes to implement in hardware. The vector is described by the first (component) word location in memory and the vector length in words. All the supercomputers considered here can handle this type of storage. The next level of complexity is regularly stored vectors. In this case the vector is described by the first word location, length and the stride (the number of contiguous storage locations between the components of the vector plus one.) Regularly stored vectors occur frequently in matrix manipulations. For example, the common way to store two-dimensional matrices is by columns. That is, each column of the matrix is successively and contiguously stored in memory. In this case, the row of a column stored matrix of order N, is a vector that is regularly stored in memory with stride of N. Contiguous storage is a special case of regular storage with a stride of one. Most of the supercomputers allow for regular storage. Those that don't are handicapped in certain situations. The random storage of vector data as indicated in Figure 2-21 clearly requires a mechanism for describing the location of the desired components. The vector is usually described by the first word location and an index array holding the relative memory location (relative to the first word location) of the remaining components. For the randomly stored vector in Figure 2-21, the first five values of an index array, INDEX (I), would be

INDEX (1) = 1

INDEX (2) = 2

INDEX (3) = 6

INDEX (4) = 8

INDEX (5) = 9.

The index array is as long as the array of non-zero data. Th: fetching/storing of a randomly stored vector data to a contiguously stored memory block, via the "mapping" prescribed by the index array, is often called the "gather" operation. The inverse operation is called the "scatter" operation. This storage scheme is quite useful when describing data related to sparse matrix data where the non-zero elements of a vector are manipulated according to an index array. (See Chapter 3.) Many of the key differences in supercomputer performance are related to the efficient manipulation of these three types of vector data. The following table gives a listing of the types of data supported by each machine. Those machines supporting random vector storage all support regular and contiguous storage. Those supporting regular storage also support contiguous storage. ("Support" is taken to mean that the instruction set supports vector operations with the particular data type.)

Allowable Vector Types

COMPUTER	ALLOWED VECTOR TYPE
CRAY-1	regular
CRAY X-MP	regular
new* X-MPs	random
CRAY-2	random
CYBER 205	contiguous
ETA-10	contiguous
FUJ. VP 200	random
FUJ. VP 400	random
HIT. S-820	random
IBM 3090/VF	random+
NEC SX-2	random

*The original X-MPs did not allow random storage. The first X-MPs delivered with random vector storage were the multiple CPU machines beginning in 1983, later the single CPU machines were all produced with this feature.

+The 3090/VP allows for this type of data but performance can degrade due to more frequent cache hits.

TABLE 2-10

The complexity of vector instructions vary from computer to computer. Currently, most of the supercomputers available require the vector input data to be independent of the output data. Thus, frequently used loops that involve recurrence are not "vectorizable". (This latter term refers to the translation of Fortran code into vector instructions by hand or compiler.) Consequently, the following loop is not vectorizable.

```
      DO 10 I = 1, N
10    Y(I) = Y(I-1) + X(I)
```

This is not absolutely necessary. With effort architectures that are basically vector pipelined in orientation could be enhanced to implement even these loops. There is one computer that can implement this loop (with some degradation over a non-recurred loop) with certain restrictions on the "delay" stride. In the above example, the delay stride is one.) The NEC SX-2 calls the loop above a first order iteration (referring to the unit delay). Unfortunately, the rate of this loop is only 2 to 4 times faster than scalar loops, but it is a step in the right direction.

This, in fact, is another first for the Japanese supercomputer community. They were the first to bring gather/scatter operations to the market, and to our knowledge NEC is the first to bring first order recurrence to the market. As vector instruction sets are enriched and enhanced, the flexibility to achieve a higher percentage of the maximum performance is increased. This also increases the complexity of software development, particularly for a multiple vendor environment.

Current trends in computer design are moving toward parallelism. One could predict, however, that if there is another instruction breakthrough in the single vector CPU, it would be to allow certain types of recurrence, (e.g. constant delay recurrence with restrictions on the delay range.)

There are many types of vector instructions, but generally they can be divided into the following categories:

1. DYADIC VECTOR OPERATIONS: *, +, /
2. TRIADIC VECTOR OPERATIONS: e.g. SAXPY
3. LOGICAL/MASK VECTOR OPERATIONS
4. VECTOR TO SCALAR: collapsed sums
   min/max of a vector
   multiply accumulate
5. LOAD/STORE
6. GATHER/SCATTER
7. BIT/SHIFT MANIPULATION
8. FIRST ORDER RECURRENCE

When a computer doesn't have a certain type of vector operation in its instruction set, it can degrade performance. For example, the SAXPY is a simple triad operation. Without it, a computer may have to resort to two dyad operations, thereby cutting performance asymptotically by a factor of two. In the case of a computer that can chain or link the result of a dyadic process directly into the next pipeline input, the degradation is confined to increased startup time, while doubling the asymptotic rate. Another example of the impact of instruction processing is the introduction of new types of vector processes. For example, if a computer cannot process vector gather/scatter operations, the process would have to be carried out using scalar instructions at a pronounced reduction in performance--limited by the speed of scalar instruction processing rather than asymptotic pipeline rates.

One type of operation in the above list is worthy of further discussion, masked operations. Most of the supercomputers available provide for bit-control vectors. These are logical vectors whose components are single bits which are either one or zero (i.e. on or off, true or false.) They can be generated by vector logical compare operations and can be used to take action according to the logical bit. For example, on a CYBER 205 one could perform a vector addition according to a bit control vector such that the result is suppressed if the corresponding bit is false. It should be emphasized that such operations are not done any faster than if all the operations were being performed. For this reason, the use of masks and bit control vectors is rare in, say, the manipulation of very sparse vectors. They are more effective for "almost dense" operations, i.e., those that have few exceptions, or in operations involving merging of vector data according to some rule.

The following table lists the categories of operations supported by various computer types. An "X" indicates the category is supported at maximum vector rate. An "R" indicates the feature is supported in vector mode, but at a vector rate significantly slower than the peak vector rate for a dense operation. A "-" indicates the operation is not supported by the vector instruction set.

Vector Instructions by Category

	DYADS (WITH STRIDE)	TRIADS (WITH STRIDE)	MASKED VECTOR	VECTOR TO SCALAR	LOAD/STORE (WITH STRIDE)	GATHER/SCATTER (RANDOM)	BIT/SHIFT MANIPULATION	1st ORDER RECURRENCE
COMPUTER :	1	2	3	4	5	6	7	8
CRAY-1 :	X	-	X	R	X	-	-	-
CRAY X-MP :	X	X	X	-	X	-	-	-
new* X-MPs :	X	X	X	-	X	X	-	-
CRAY-2 :	X	-	X	-	X	R	-	-
CYBER 205 :	R	R	X	R	X	R	-	-
ETA-10 :	X	X	X	R	R	R	R	-
FUJ. VP 200 :	X	X	X	X	X	X	X	-
FUJ. VP 400 :	X	X	X	X	X	X	X	-
HIT. S-820 :	X	X	X	X	X	X	X	-
IBM 3090/VF :	X	X	X	X	X	R	X	-
NEC SX-2 :	X	X	X	X	X	X	X	R

TABLE 2-11

A final topic, related to control and instruction processing, is the relationship between the processing of scalar and vector operations. The most restrictive form of instruction processing is to process only one instruction at a time. This is not unusual on scalar machines. On vector computers, however, while a long-vector operation is executing, a considerable amount of scalar instruction processing could take place, ignoring possible memory conflicts. It is also very common to have a number of vector instructions linked together so that at any one time a number of vector instructions are in simultaneous execution. Instruction overlap, of this kind, is easier to achieve on some machines than others. The variations are dependent on each instruction as well as the architecture. For example, on a CRAY series computer scalar floating point operations and vector floating point operations use the same functional units; therefore, they cannot be overlapped. In contrast, the NEC SX-2 provides separate scalar and vector units, and as long as data contention is avoided, scalar and vector operations can proceed in an overlapped fashion.

************* **IMPORTANT CONCEPTS** *************
Instruction Processing

- Computers that manipulate "vectors" deal with restrictive definitions of mathematical vectors. These vectors are defined by memory storage characteristics that can vary from machine to machine. The three main types of vector storage are listed below. (Note: each is a subset of its successor.)
    - contiguously stored (first word location and length)
    - regularly stored (first word location, length, and stride)
    - randomly stored (first word location, index or pointer array)
- One common rule pervades vector operations: the input vectors must be independent of the output vector. Exceptions are beginning to appear in computer designs (e.g. NEC SX-2), but the principle remains fundamental in high performance algorithm design.
- Support of categories of vector operations varies among supercomputers. It is important not only to know whether an instruction type is supported, but if it operates at maximum rates or not.

****

Secondary Storage

Figure 2-12 displays a storage medium between main memory and conventional disks. This device is secondary storage and is employed by several supercomputer manufacturers to both improve performance and reduce the price of large memory systems. What has become a glaring problem is the inability of conventional disk technology to keep pace with the rapid growth in computational power and central memory of modern computers. The industry has witnessed almost geometrical growth in computing power over the past several decades, with little sign of abatement. The growth in disk technology with respect to speed of access and capacity, has been more arithmetic in nature. The growing disparity between what we can compute and the availability of storage to hold the results has become a major computing issue.

The basic fact is that only disks can permanently hold problem input and the resulting output data. Unfortunately, modern scientific computations often generate much greater amounts of intermediate calculations which typically must reside in memory. When available memory fills, a number of alternatives exist for moving data in and out of inadequately sized memory. A common scheme in conventional computing for management of this data is virtual memory. With VM, the operating system has the task of "paging" data in and out of memory to a disk or other storage device. A more common method among supercomputers is to have the user manage the data movement directly with I/O instructions accessible in the Fortran language. As the disk storage characteristics have become so inadequate relative to computing power and main memory speeds, the need for an intermediary storage device becomes apparent.

As an example, the CRAY X-MP system can include a Solid-state Storage (SSD). Its characteristics are quite impressive relative to disk technology. At the time of its introduction, the SSD single word access time was ten times faster than disk technology, but with a bandwidth improvement of over 250 times that of conventional disk, for large amounts of data. The speed of the SSD is almost that of single ported memory for large amounts of data. As a storage medium, secondary storage lacks a basic feature, permanence. Secondary storage is volatile memory, and cannot be used to store data permanently. The advantages main memory has (on an X-MP) is 3-paths and less initial fetch overhead. In Chapter 3, examples will be given of how SSD-like devices can improve performance.

Table 2-12 lists supercomputers that employ some form of secondary storage to improve throughput. The table lists the size of secondary storage in 64-bit words, the memory bandwidth and secondary storage bandwidth. Also included in the table is an entry related to disk striping, the ability to utilize parallel'sm in I/O to conventional disk. Disk striping is nothing more than using parallel channels to send a file to multiple disks simultaneously. This is accomplished at the operating system level where the data is split in a systematic fashion and sent to multiple disks (a sort of striping of the data occurs, where each stripe is assigned to a disk and can be recovered in a similar fashion.) The advantage is a simple bandwidth improvement proportionate to the number of disks involved in the striping. For example, the NEC SX-2 allows 32-way striping. This in turn can improve I/O by a factor of 32 for large amounts of data.

Use of Secondary Storage

COMPUTER	SECONDARY STORAGE	DISK STRIPING	MEMORY BANDWIDTH	2ND MEM. BANDWIDTH
CRAY X-MP	512 MW	yes	352 MW/SEC/CPU	125 MW/CHANNEL
CRAY-2	none	yes	750 MW/SEC/CPU	N/A
ETA-10/E	256 MW(main)	yes	285 MW/SEC/CPU	95 MW/SEC/CPU
FUJITSU	none	yes	533 MW/SEC	N/A
HIT. S-810	384 MW	yes	286 MW/SEC	125 MW/SEC
IBM 3090	64 MW	yes	N/A	162 MW/SEC
NEC SX-2	256 MW	yes	1300 MW/SEC	162 MW/SEC

TABLE 2-12

The growth in main memory sizes poses some interesting design questions. Does a computer manufacturer strive for massive memories and/or a tiered memory structure. From the view point of the functional units, memory is already quite structured. Memory options include the following:

Volatile storage-

- Registers
- Cache
- Central Memory
- Secondary Storage
- Tertiary Storage (not very common)

Permanent -

- Disks (striped and regular)
- Tapes
- Massive Storage media

It is typical for a device higher on the list than a lower device to have faster access times for single pieces of data, greater bandwidth, and lessor size (in terms of words). These devices can be viewed as the palette and the architect the artist. In the design of computers, all or some of these devices can be employed in the development of the memory hierarchy. A very current example of the dilemma of design of memory is depicted by the interesting differences offered by Cray Research. As of this writing, the CRAY-2 is the largest single main memory computer available. When the first CRAY-2 was delivered in 1986, it represented more main memory than all the Cray computers delivered prior, combined. Such an astronomical change in computer balance, can redefine the computational process. The CRAY-2 has no secondary storage option -- simply huge memory. Which design (SSD or large main memory) is better? The question is moot. A better question is for which computer design will more applications be productive. Today at NASA Ames for example, there are jobs that run slower on the CRAY-2 than the X-MP. At the same time, there are models being run on the CRAY-2, that cannot be attempted on the X-MP because of memory limitations (even with a large SSD).

Supercomputer power is a balance of speed and memory. The formula for a successful combination is not known and, at best, is problem dependent. The success of a computer design is measured by its economic usefulness in computation. Radically new designs demand radically new approaches to computation. The CRAY-2 is very much an experiment in a new balance in computing speed and memory. Even with a full complement of disks, the CRAY-2's memory cannot be "emptied" to disk in the same time frame as a regular supercomputer. Consider the operating system changes that must be considered. Job swapping, virtual memory, the disk relation to main memory, algorithmic approaches to problem decomposition all need redefinition. The impact of such a massive increase in memory, must be accommodated over the entire spectrum, from operating system to user model.

This discussion would not be complete without noting another aspect of I/O -- channel speeds. In large application programs permanent output data is being gradually replaced by a graphical display medium which is perhaps interfaced to the computer by another CPU. This approach is putting an ever greater demand on supercomputer's channel speeds. Table 2-13 shows that supercomputers can differ in this important throughput characteristic. The greater the channel speed, the greater the flexibility in "integrating" a supercomputer into a usable and productive system. There are those who consider channel speed as important as computational speed and memory size when rating computer "power".

Supercomputer Channel Speeds

COMPUTER	CHANNEL SPEED(/sec)
CRAY X-MP	224 Mbytes
CRAY-2	4000 Mbytes
ETA-10/E	250 Mbytes
FUJITSU	96 Mbytes
HIT. S-810	96 Mbytes
IBM 3090	96 Mbytes
NEC SX-2	96 Mbytes

TABLE 2-13

2.2.4 The Use of Multiple CPUs in Supercomputer Design

The trend to increase the computational power of Supercomputers through replication of CPUs, is well established. CRI, IBM, and ETA Systems offer multiple CPU machines. The initial foray into this arena has been largely motivated by job stream throughput considerations. Two or four CPUs, at best, can only increase performance by a factor of two or four respectively. This can only be accomplished through some modifications to the application at hand, usually by the user. This requires time and effort. On the other hand, providing two CPUs, in proper coordination through the operating system, can easily, and without user intervention, nearly double the throughput of a system. For this purpose, IBM and its plug compatible competitors have been offering multiple CPU mainframes for some time in scalar computers. The trend in supercomputing, however, is now proceeding toward 8 or 16 CPUs. Cray Research will be offering 8 and 16 CPU machines on the X-MP, the Y-MP, and/or the CRAY-3 by 1987/8. Several trade journals have predicted 8 CPU 3090s in the future as well. When the number of CPUs exceeds 4, the gains from improving throughput begin to wane. Eight job streams would tend to randomly compete for critical resources such as memory and I/O ports to the detriment of performance. To effectively use higher degrees of parallelism, the power of multiple CPUs has to be brought to bear on a single job. In order for this to take place the application program must be redesigned, and the operating system has to provide the mechanisms for higher level languages (such as Fortran) to create and coordinate tasks within a single job. Through this coordination (synchronization) the programmer can perhaps eliminate random contention and create much greater "job" throughput. Ideally, if N processors are brought to bear on a single job, a performance improvement of a factor of N in real time is attained. Invariably, not all the contention for resources can be eliminated. In addition, the CPU coordination itself introduces overhead that can degrade potential performance. Software performance issues of this kind will be discussed in Chapter 3. In this section, different hardware philosophies of parallelism being employed by the three vendors mentioned will be briefly discussed.

Among the three vendors alluded to above, Cray Research was the first to offer a multiple CPU system with software to support the synchronization and coordination of these CPUs within a single job. (CRI has called this process multitasking. In addition, CRI provides other more elemental tools for coordination and synchronization which they have chosen to call microtasking. While these terms are specific, the community seems to be adopting the term "multitasking" as a vendor generic term. In this treatise we will use the term in the generic sense unless otherwise indicated. See "multitasking" in the glossary for more elaboration.) The software aspects of multitasking will not be discussed in this section, rather an overview of the design will be described.

The first multiple CPU machine from CRI was the CRAY X-MP. The simplest form and structure for parallelism was employed. Figure 2-22 gives a conceptual overview of this structure which applies to the CRAY X-MP and the CRAY-2 as well. The key feature is that the CPUs share (and thus contend for) main memory. Each CPU is autonomous in terms of operation. Thus, with the proper software true MIMD processing can be implemented. When one or more CPUs require data they must share access to memory with a hardware/operating system convention for breaking ties. A number of anomalies can occur with regards to memory bank and memory section conflicts. (For a complete treatise on this subject for the X-MP see [2-12].) The nature of the conflicts extend beyond the simple contention for identical data because of the bank organization of the CRAY computers. Each time a bank is accessed there is delay time required before it can be accessed again. Any attempt to access the bank before the wait time has elapsed will cause a suspension of activity. The greater the wait time in the hardware design, the greater the penalty. Thus, a good program design must account for data distribution in memory across banks, as well as analytical contention for identical data. Table 2-7, in Section 2.2 gives the bank wait times in cycles. Probably the most notable wait time is with the CRAY-2, where even though it has a large number of banks, the over 50-cycle wait time can become a problem with only one CPU, let alone four.

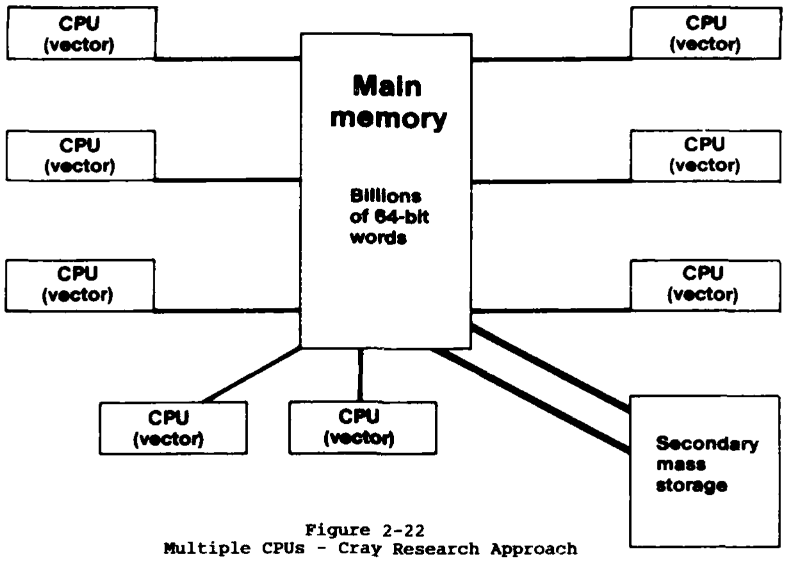


Figure 2-22
Multiple CPUs - Cray Research Approach

The description of a single CPU for an X-MP or a CRAY-2 (that was presented in the previous section) applies to multiple CPU systems. The only significant addition in the multiple CPU case is a description of CPU coordination registers. This feature allows true MIMD processing. Each CPU has semiphore register for communication among the CPUs. Thus, a CPU can be directed to take actions based on the status of the semiphore bits available to all CPUs. In this fashion, coordination of CPUs can be achieved. CRI has utilized this feature to enhance Fortran with multitasking capabilities through operating systems routines and also to support *"microtasking"* compiler directives. These techniques will be briefly discussed in Chapter 3.

In contrast to the CRI approach, ETA Systems has adopted a different approach. Figure 2-23 gives a conceptual overview of their machine. The striking difference is the use of memory. Currently, each CPU has 4 MW of memory. The central main memory, from the frame of reference of a single CPU, is much like secondary storage on an X-MP. In fact, the bandwidth from a single CPU is roughly the same as the bandwidth from an X-MP to its secondary storage device. In addition, the design provides for inter-CPU communication through a million word buffer. Thus, in the extreme cases, this design trades memory contention of the CRAY design for communication contention among the CPUs. That is, if one CPU needs to compute data generated by one or more of the other CPUs, a communication overhead is incurred.

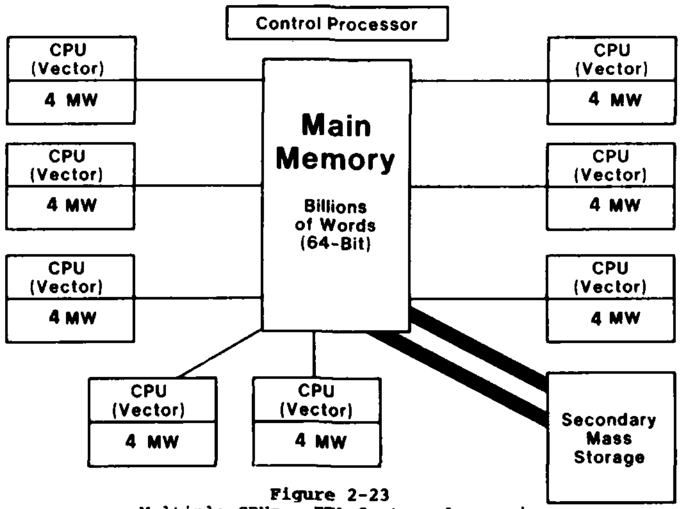


Figure 2-23
Multiple CPUs - ETA Systems Approach

The IBM six-CPU machine is the 3090/600. The most powerful version would have a Vector Facility (VF) with each CPU. While the interface of the CPUs to main memory is a bit more complex than the CRAY series of designs, it basically employs the same philosophy of the CRAY X-MP. It is quite conceivable that there are many parallel computational schemes for algorithm or application design that would produce efficient use of both the CRAY and IBM architectures. In another words, the possibility of transportable parallel programs between the 3090/400 and CRAY X-MP series is not out of the realm of possibility. This would be more difficult for an ETA Systems machine.

************* **IMPORTANT CONCEPTS** *************
Multiple CPUs

- Supercomputer designs are moving toward relatively small degrees of parallelism with each CPU being a pipelined powerful machine in its own right.

- Various scenarios will appear
  - shared memory, perhaps tiered (this can cause memory contention among CPUs)
  - distributed memory, perhaps tiered (this can cause communication overhead among CPUs)

****

2.3 EMERGING TECHNOLOGIES

2.3.1 Introduction and Intent

It is the purpose of this treatise to discuss the current generation of supercomputers and their relation to computational fluid dynamics. The question often arises as to what is next. Vendors are often very closed mouthed about new products. In the supercomputer arena, however, the nature of new products is not just kept secret, it is a basic research topic on which company survival is built. The ideas for new architectures and underlying chip technologies are obtained by vendors from many sources. In order to anticipate tomorrows designs it is often fruitful to look at today's mid-range computing trends and research projects in academic computing environments. To survey parallel computing architectures or even the new mid-range supercomputers (near-supercomputers) is a formidable task, indeed. Currently, there are over 20 vendors in the market with some form of parallel and/or vector computer in the minicomputer class price range, and over 50 companies with designs seeking funding to bring products to market. The likelihood that more than 5 or 6 companies will survive, over the long term, is small. In addition, the traditional minicomputer companies such as Digital Equipment are one by one entering this high performance, low cost scientific computer market with parallel/vector architectures. In this concluding section to Chapter 2, we will briefly describe the various forms of parallel architectures available today, and examine (using the CPU model in Figure 2-12) several minisupercomputers, namely, ALLIANT, CONVEX, FPS 164/264, and SCS-40.

2.3.2 Parallel Systems

A complete taxonomy of parallel architectures will be avoided. Nevertheless, it is worthwhile to examine several classes of parallel architectures, recognizing that many machines are combinations of attributes from several classes. Probably the most basic distinction to be made is "shared" vs. "distributed" memory systems. This distinction has already found its way into supercomputer design. As mentioned earlier the CRI multiple CPU machines are basically a shared memory systems, wherein each CPU has equal access to, and hence contends with other CPUs, for main memory. The ETA Systems GF-10, on the other hand, distributes primary memory offering a shared secondary memory. (See Figures 2-22 and 2-23.) The two approaches are fundamentally different.

In the smaller machine market, the CPUs themselves don't have the power of a supercomputer CPU. Quite often the manufacturers provide a greater number of CPUs which, in the shared memory case, contend for access to main memory through a high speed bus, switching network, and/ or layered caches. Table 2-14 is a sample of shared memory machines. Unfortunately, many machines defy classification by combining features of several basic designs. For example, some of the shared memory machines also allow CPUs with local memory. The decision to classify them as shared versus distributed is somewhat arbitrary.

Shared Memory Machines

Company - Machine	No. CPUs	Communication Type	Memory
Alliant - FX Series	1-8	cache	shared
BBN - Butterfly	2-256	switched net	shared/local
Concurrent 3280	1-6	bus	shared
Elxsi	1-12	bus	shared/cache
Encore	2-20	bus	shared/local
Flexible	1-20480	bus	shared/local
FPS/164-MAX	4-16	bus	shared
Masscomp 5700	1-4	bus	shared
Sequent - Balance	2-30	bus	shared

TABLE 2-14

In distributed memory systems each CPU has its own memory. In turn these CPUs are linked by a variety of connection schemes. The CPUs can be ringed, linear, latticed or connected in a variety of ways. Ringed or clustering of CPUs seems to be a popular approach to utilizing more powerful CPUs with substantial memory. Typically one finds that the more powerful the individual CPUs, the smaller the implemented degree of parallelism. It also follows that the fewer the number of CPUs, the less sophisticated the connection scheme. The most popular distributed architecture from a commercial point of view is the hypercube architecture. (For a description of hypercube architecture, see the glossary.) There are already five commercial companies in the market place with hypercube machines.

The most amazing aspect of commercially available hypercubes, is the variety of approaches. These differences are so great that the entire approach to algorithm design for effective utilization of the hardware can change character among machines in this single class. For example, the maximum number of CPUs available from a given vendor, range from 128 to 64000. The computational power of each vendor's single CPU ranges from bit oriented processors (the Connection Machine) to full floating point 20 MFLOPS/CPU. As indicated earlier, the more power the CPU, the fewer the number of CPUs available. In addition, there are very big differences in memory sizes and communication speeds between CPUs. If one, for example, has a Floating Point Syste T-Series hypercube with 16 CPUs, one's approach to computation would be considerably different than for the 64000-CPU Connection Machine. In the latter case, one could view the machine as one big semi-programmable memory (in the spirit of an associative processor).

In any distributed processor design, certain key parameters dictate the nature of software design. The parameters become the menu for the architect in an exercise of compromise. Table 2-15 lists some important considerations. While this is not meant to be a comprehensive discussion of parallel architectures or of issues in their design, it is instructive to examine the various differences observed in commercially available machines. This is given in Table 2-16. It is interesting to point out that the largest number of CPUs is available in the Connection Machine where individual CPU power is the lowest. It is also worth mentioning, that the next model of the Connection Machine will have more computational power by introducing a computational floating point chip to be shared by every 32 CPUs. One can imagine no end to the variety possible in the world of parallel designs!

Parameters in a Distributed Environment

Parameter	Considerations
CPUs	The number of CPUs The computational power Internal I/O overhead
Memory	The size of individual memory The availability/addressability of secondary memory or other processors' memory
Network	The connection scheme (full or partial: e.g., ring, cube, clusters) Shared or switched busses
Bandwidth	Communication rates between CPUs (near and far neighbors)
Control	Instruction processes and timings (broadcast and local) Synchronization primitives

TABLE 2-15

Hypercubes

Manufacturer	No. CPUs	Local Memory (bytes)	CPU power (Mflops)	Bandwidth (Mbytes/sec)
FPS T-Series	8-16384	1000K	15.0	7.5
Intel iPSC	16-128	512K	0.1	10
NCube	4-1024	128K	0.3	180
The Connection	16K-64K	0.5K	10**(-3)	

TABLE 2-16

It would be remiss to exclude from the discussion a listing of distributed memory systems not in a hypercube configuration. For example, there are systems that have networks in nets or mesh configurations. These may have direct applicability to CFD algorithms and other partial differential equation problems. Like hypercubes, such machines are quite varied in approach, CPU power, and memory. Again, Table 2-17 lists a sample of machines in this category. The three machines are very different. The Cyber Plus is a ring connected set of up to 16 CPUs with several data memories and an instruction memory. Up to 16 rings can be connected to a Cyber 800 host. Each of the CPUs can operate independently, each has a 20 nanosecond clock cycle. Since the main data memories can have as much as a half a million 64-bit words, each single CPU is a rather substantial machine. The ICL DAP, on the other hand, is a SIMD CPU device with up to 4096 CPUs arranged in a lattice or mesh. The Goodyear MPP has bit serial CPUs in a lattice or mesh structure with edge closure (i.e. the mesh edges, left and right, top and bottom are nearest neighbors.)

Distributed Memory Systems

Company - Machine	No CPUs	Communication Type
CDC Cyber Plus	1-16	ring
Goodyear MPP	16K	mesh
ICL DAP	32*32 or 64*64	mesh

TABLE 2-17

The intent of this all-to-brief survey of parallel processors is to give the reader an idea of possible directions that supercomputer manufacturers may take with future designs involving multiple CPUs. Cray Research and ETA Systems are already implementing different, yet simple, forms of parallelism. Academic institutions, government laboratories, and some large established computer companies (e.g. IBM) have prototypical parallel machines which we have not covered, but they all share some of the ideas of designs presented.

************* **IMPORTANT CONCEPTS** *************
Parallel Systems

o It is common to see new machines with parallel CPUs. Typically the more powerful the individual CPU, the fewer there are available in the design.

o A popular commercially available architecture is the hypercube. Three characteristics of distributed memory machines that determine performance and algorithm design are: the computational power of individual CPUs, the bandwidth between CPUs, and the memory capacity at each node. The smaller hypercubes are message passing systems, and the overhead of sending a message also becomes important in algorithm design considerations.

o As the number of CPUs increases, the nature of algorithm design is dramatically affected.

****

2.3.3 Minisupercomputers: A Market Perspective

Thus far the topic of "minisupercomputers" has been avoided. The term minisupercomputer is defined in the glossary, Appendix A. The definition is given in terms of a performance gap between supercomputers themselves and popular minicomputers. In the next section, the four leading competitors in this price performance "gap" region will be discussed. They are the Alliant FX-8, Convex C-1, Floating Point Systems FPS-264, and Scientific Computer Systems SCS-40. These machines are all very different in their design, but share attributes of computers already discussed. Before discussing these machines, it is worthy to discuss the minisupercomputer phenomenon in non-technical terms from a market perspective.

It is generally acknowledged that currently the market for supercomputing has grown tremendously. Some of the supercomputer vendors are enjoying moderate to excellent success in the market place. However, it is generally conceded by most observers that the supercomputer market is a fragile one from an economic perspective. To complicate the picture a number of new and emerging computer vendors are bringing a wide variety of products into the marketplace. There are claims of devices being "supercomputers on a desk", "personal supercomputers", "graphics supercomputers", and "poor-man's supercomputers". Recently, an article in one of the trade journals claimed that a large aerospace company was replacing their supercomputer with a number of these new devices. (It was later denied as ridiculous by the named company.) This relative confusion as to the role of these new "minisupercomputers" can create havoc in this already "fragile" industry.

The Setting

Minisupercomputers are both a technical and marketing phenomenon. They are more easily described in terms of what attributes they have relative to the more traditional machines in the market. We confine our discussion to the engineering and scientific computing marketplace, since this is where the "action" is relative to these machines. We will begin with the following definition from the Glossary.

> SUPERCOMPUTER(S): The class of general purpose computers that are both the fastest available commercial machines AND have sufficient central memory to store the problem sets for which they are designed. The issue of "computer power" in large-scale scientific processing is a complex topic. Computer memory, throughput, computational rates, and a host of related computer attributes contribute to performance. Consequently, a quantitative measure of computer power does not exist, and a precise definition of supercomputers is difficult.

From a technical point of view, supercomputers combine breakthroughs in architectural design, chip technology and/or packaging to achieve very significant improvement in performance. In order to do this, supercomputer companies must remain on the very fringe of research and technology. Often prototypes of early machines attempt new approaches to circuit cooling and packaging. The newest models often have immature software systems due to the inability of the architect to be both innovative and compatible to previous models. These companies must put a tremendous amount of investment in research and experimentation. The result, if successful, is a machine that is so fast that the expense is more that recovered in speed. This invariably results in a price/performance breakthrough.

From a market perspective, the supercomputer is a bit of a miracle. By their very nature supercomputers are so expensive that only very large companies and/or organizations can afford to purchase one. In other words, price performance is only an advantage to those who can afford to "buy-in". Furthermore, being able to afford such a machine alone is rarely enough incentive to purchase such a device. Most supercomputer owners have another attribute. They have problems that cannot be solved on conventional machines. It is not simply a matter of quicker turn around. These companies have users who are typically advancing the state of science and leading the way toward improved product designs (e.g. aerospace industry) or improved processes (e.g. petrochemical industry).

The computer market seems to operate under an economic law that is difficult to make precise. We will give it a name, and try to define it below:

> THE LAW OF PRICE/PERFORMANCE CONTINUUM: The computer market tends to push for new levels of price performance. Once a new level of price/performance is achieved, economic realities tend to fill in a continuum of product choices. Portions of the market are willing to pay for high performance (small volume business) and others want less performance for a smaller price (higher volume business).

Historical Examples

For years large mainframe computers competed on a performance basis. Every few years, largely through improved chip performance, more powerful machines came to the market with the same or even smaller price. As the performance improved, a gap appeared at the lower performance end of the spectrum. It was recognized that by relaxing performance, cheaper air-cooled machines could be made easily with mass-produced chip technology. They had all the functionality of the high-end machines at equal or better price/performance. This gave birth to the minicomputer industry.

The process repeated itself more recently. As the minicomputer industry chased the high end, a new gap opened. This was filled by the microprocessor, and the personal computer revolution struck with astounding results.

In the above examples, one important aspect has been ignored -- software. In all instances, the gap fillers were not truly successful until they came to the market WITH usable software systems. For the minicomputers, they not only were able to run mainframe software, but provided software capabilities in terms of interactive computing and graphics that their "big brothers" had not accommodated. Similarly, the microprocessors brought word processing, spread sheets, and similar capabilities that helped define a "reason" for the new price/performance niche beyond simple economics.

The Minisupercomputer Phenomenon

With the above as background, one can see that the minisupercomputer phenomenon is less a phenomenon and more the natural course of economic evolution in the computer industry. It begins with a new and substantial breakthrough in high-end performance (as well as price performance). After suitable moratorium period, required to let this market mature with customers and software, the gap is formed. Based on unprecedented growth of successful supercomputer company(ies), even during sluggish economic growth in the computer industry, a well-defined customer set with high performance requirements emerged. A multi-billion dollar market is, in fact, suspected.

The market approach, to fill the gap between supers and minis, almost defined itself. Clearly, one could not continue to push chip technology and gain the kind of price/performance supercomputers have achieved. So, simply building a faster mainframe or minicomputer would not "hack it." Supercomputer manufacturers, in fact, had achieved the improved price performance through "parallelism" in the form of pipelining and multiple CPU hardware. The solution was simple: use the chip technology and manufacturing techniques of minicomputer manufacturers and the computer architectures of modern supercomputers. The latter architectures employ "vector" hardware and/or parallel CPUs to achieve better throughput in scientific processing.

The early minisupercomputer manufacturers are working very hard to achieve software parity with the supercomputers at the high end and the minisupercomputers at the low end. If, in fact, they can achieve software stability, they can challenge both ends of the market for a unique niche. As in the examples above, it is also likely that, in time, this class of machine will define new software domains. Some predict that economics and business applications may be amenable to this class of equipment from a price performance point of view (but currently vector software in this arena is missing.) Can they drive either end out of business? If history serves, this is not only unlikely, but counter productive for all concerned, as discussed further below.

The Impact of Minisupercomputers

It would take a very large crystal ball to foresee the total impact of minisupercomputers. Who would have ever imagined the impact of personal computers on society. Could Henry Ford have predicted the 8 lane freeway when he built his first car? One underlying fact seems to pervade this whole discussion. The scientific community has a fundamental need for more computing power. This seems to be true throughout the spectrum of computing. More power is required on the desk, in the office, in the departmental computer center, in the company's data center, and so on. Productivity would dictate that interaction, editing, quick turn-around, debugging runs, etc., would require a certain amount of intimacy between the user and the computer. Requirements for large aerodynamic design, chemical analysis, structure design, and a host of various industries' leading modeling challenges, will require the enormity of computing power available only from pioneers in high-performance processing. It is likely that new gaps in performance will appear. As we see today's minicomputer performing computations only mainframes could achieve a few years ago, it is likely we will see today's supercomputer load easily accomplished on tomorrow's minisupercomputer, and, if history serves, a new software requirement will be the domain of even more powerful true high-end supercomputers. At the same time we can observe new workstation technology, defining new ways for scientists to interact with their models. For example, rather than computing pressure distributions on a wing to infer lift, today's aerodynamicist is "watching" the flow (color coded for pressure) over the wing through complex graphical displays.

Historically, new popular "niches" in the computing spectrum have taken-on their own character. In time minisupercomputers will find their own community of users. It is clear, however, that these machines will be a bridge between the low end and high end of the computer industry. The bridging will be through various approaches to software/system compatibility. For some, the minisupercomputer will be the local supercomputer. That is, play the role of the mainframe in a smaller company or organization. For the large system companies, current owners of supercomputers, the minisupercomputer will fill a different role, that of secondary and off-load development. Minisupercomputers for this group will perform duties related to graphics, interactive development, and departmental computing. The four active contenders in this arena take on their own characteristic flavor in terms of market position. While no official position is taken by the companies, observation leads one to the following description:

> **Alliant** - focused on UNIX and parallelism, Alliant has had initial success in the research community. With this as a base, the company is striving to create its own community of users with a growing number of application packages. Other than UNIX, no initial thrust was made in compatibility with minisupercomputers or supercomputers.

Convex - Convex has strong desire to be the compatible upgrade to the VAX scientific user community. Great effort has been placed in VAX compatibility (with varying degree of success). A strong program to add application programs positions itself as a potential stand alone computing engine. Many claims of Cray compatibility through Fortran have been stated based on the aggressive vectorizing compiler. The compatibility, however, is based on software development methodology and has no strong basis in architecture or underlying assembler compatibility.

FPS-264 - Floating Point Systems, Inc., is a well established computing firm. Its history is based in the array processing field where it has virtually dominated for years. The FPS-264 is very similar to the companies initial product, with the addition of true instruction memory supporting Fortran. Many of the remnant philosophies of an array processor company have plagued FPS's positioning as a minisupercomputer. Recently, however, the company has made a strong market push for identity as a minisupercomputer. They have a large customer base in providing add-on equipment for VAX owners and are likely to pursue this market in direct competition with Convex and SCS.

SCS-40 - Scientific Computer Systems, Inc. was founded on the principle of Cray X-MP compatibility. The SCS-40 faithfully emulates the X-MP assembler instruction set. As such it can run both X-MP's popular operating systems, COS and CTSS and most applications software that run on those systems. It is clearly positioned strongly as a peripheral node to a CRAY X-MP environment. It can be utilized as an off-line development machine or as the first step toward ownership of an X-MP. The company is also striving to achieve VAX software compatibility in the sense of Convex, and thereby, challenge both FPS and Convex in that market.

************* **IMPORTANT CONCEPTS** *************
Minisupercomputers

o As the high end computers stretch the performance spectrum, new price/performance niches will appear.

o Minisupercomputers are simply another filling-in of a price performance gap opened by technology at both ends of the computing spectrum. Other examples include, minicomputers and microcomputers.

o The impact of minisupercomputers is unclear. They no doubt will be integrated into supercomputer systems environments as well as stand alone as department computers. They will likely become the home of certain applications that have not become a part of the supercomputer scene -- perhaps data base and business applications.

o Minisupercomputers are already opening new technological environments for supercomputers themselves!

****

2.3.4 Four Minisupercomputers: Architecture Descriptions

From a computer architecture perspective, the Alliant and FPS-264 are not very similar to today's supercomputers. They were developed more "from scratch" and have employed different techniques of design. The Convex C-1 and SCS-40, on the other hand, are "true" minisupercomputers in that they are based on minicomputer hardware technology and vector pipelined architecture. The following descriptions of the four machines are purposefully brief, but serve to illustrate their similarity, or lack thereof, to supercomputers. Regardless of their architectural compatibility, each of these machines shares some similar software methodology considerations to true supercomputers. This will be discussed further in Chapter 3. The table below gives the peak rates and typical $N_{1/2}$ values for a SAXPY operation on the four machines, according to [2-11], [2-13].

Minisupercomputers: Performance Characteristics

Machine	Cycle nanosec.	Peak Rate (64-bit MFLOPS)	SAXPY MAX RATE	SAXPY $N_{1/2}$
Alliant FX-8 (Fortran)	170	94	14	150
Convex C-1 (Fortran)	100	20	10	31
FPS 264 (assembler)	54	38	35	12
SCS-40 (Fortran)	45	44	32	72

TABLE 2-18

The Alliant: The Alliant FX-8 has the distinction of having one of the most elegant vendor supplied hardware diagrams. (See Fig. 2-24.) It is simple, clear, and of adequate detail to infer the machines philosophical layout.

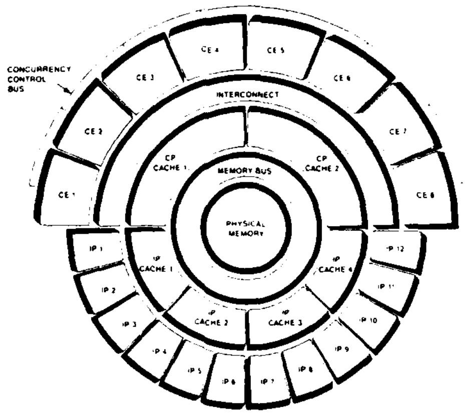


Figure 2-24
The Alliant FX-8: Hardware Description

The two major differences between the Alliant and the 5 major supercomputers, are the use of parallel CPUs and the layered cache interface. While Cray and ETA Systems have multiple CPUs, Alliant's focus was to exploit the multiple CPUs for computational power through automatic parallelization through the use of the compiler. The success of this approach is debatable, and Alliant is currently looking at alternate approaches to utilizing parallelism, particularly an MIMD approach. The use of the cache interface has software implications that are not present in non-cache systems. Algorithms must manage to confine data access within the cache whenever possible. Fig. 2-24 displays the CPU access to main memory through caches. Each layer of memory requires a longer delay to access and must be done by cache "uploads" of data. This becomes a real problem when data access causes a repeated flip-flop of data in and out of cache. Such a process could create a serious performance bottleneck. The machine compiler exploits algorithm pipelining, discussed in Section 2.1.5, as well as automatic parallelization of inner loops. This latter technique is similar to "Cray microtasking" and is a SIMD approach to parallelism. Thus, in order to achieve the peak performance listed in Table 2-16, one requires long vector operations, in a fashion similar to pipelined supercomputers. There are no hardware impediments to offering a MIMD capability which is a likely future direction. The Alliant machine is also an integral part of the University of Illinois' Cedar Project. In this project, Alliant FX-8s are being clustered to achieve higher depths of parallelism.

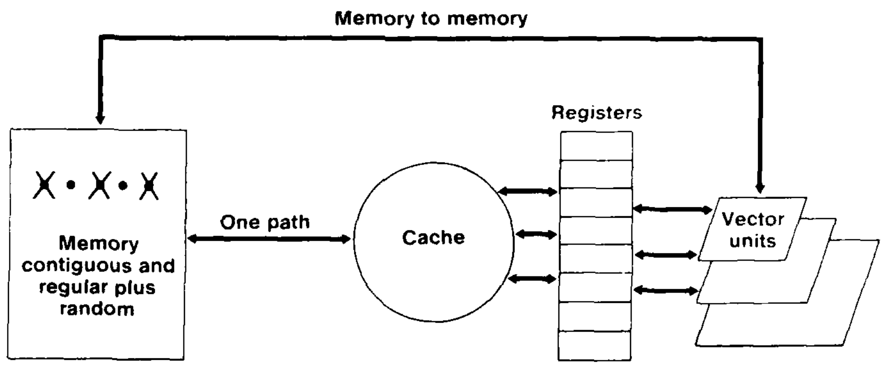


Figure 2-25
The Convex Interface

The Convex C-1: The Convex C-1 is the first true minisupercomputer from a philosophical point of view. The architect, Steve Wallach, has been very vocal concerning the concept. The idea was to marry the minicomputer technology with the reduced instruction set approach to emulate vector instructions. The result was a vector machine with a minicomputer price. The machine can be described by the supercomputer model given earlier in Fig. 2-12. Vector pipelined high performance functional units have been added to a scalar design. The interface, however, is through a cache and one memory-to-memory connection. Figure 2-25 gives a conceptual view of this interface.

The most important aspect of a vector computer design is data flow. In any vector machine an important question is: is there enough bandwidth and paths to support the necessary data flow? The Convex C-1 boasts of 60 MOPS, counting the load and store operation. In fact, this translates to 40 MFLOPS doing 32-bit arithmetic. In 64-bit floating point computation the correct peak rate shown in the Table 2-18, is 20 MFLOPS. One should always look closer at the data flow to realize other limits of performance. The bandwidth from memory to cache is 10 megawords (MW)/second (64-bit) served by a single path. In addition, the bandwidth from main memory to the functional units is 10 MW/sec, provided the data is contiguously stored. The rate of 20 MFLOPS holds for operations contained in cache and degrades otherwise. To ameliorate the complexity of this situation Convex has provided a first rate compiler by all accounts. However, the compiler cannot overcome basic peak rate limitations; rather, it can aspire to attain them over a broader band of computational algorithms or kernels.

The SCS-40: From an architectural perspective (at the level of detail that we have pursued), the SCS-40 is an identical copy of the CRAY X-MP. This is achieved with a high bandwidth internal structure between all functional units, registers, and memory. Through microcode emulation, exact X-MP instructions are performed. The internal busses have greater bandwidth relative to the computational power of the CPU than the X-MP. Unlike the Convex C-1, the SCS-40 does not appear to be a scalar minicomputer with vector capability added. It rather looks like a machine designed to be both a fast scalar and fast vector machine employing minicomputer chip technology for competitive pricing. The machine can fully support its peak performance claim of 44 MFLOPS, i.e. there is no internal bandwidth or cache limitation as with the IBM 3090 or Convex C-1. While its peak vector rate is 25% of that of the X-MP, its scalar rate is somewhat faster than that. This is in sharp contrast to say the CYBER 200 series whose scalar performance was never well balanced relative to its vector speed. The CYBERs were first designed to be vector machines and later scalar performance became an issue. Hanon Potash, the SCS architect, has often claimed that the key to designing a vector machine is to "super-impose" a scalar design and vector design to achieve the internal bandwidth to support both computing philosophies. This has the advantage of eliminating data flow bottlenecks by design. In fact, in a recent benchmark using the Argonne Labs test cases, Dongarra [2-11], this machine achieved 41 MFLOPS, or 93% of peak performance rate.

The FPS-264: This attached processor is a very interesting machine from an architectural point of view. In this treatise we have avoided a rigid classification scheme for these modern computers. Yet, the FPS-164/264 eludes any classification scheme. It is simply different. It exploits a form of parallelism that is similar to the parallelism between CPU and I/O in most conventional computers. That is, the machine is really a series of functional units that can be executed in parallel. Figure 2-26 displays the architecture of the FPS 264 in slightly more detail than we have used for other machines.

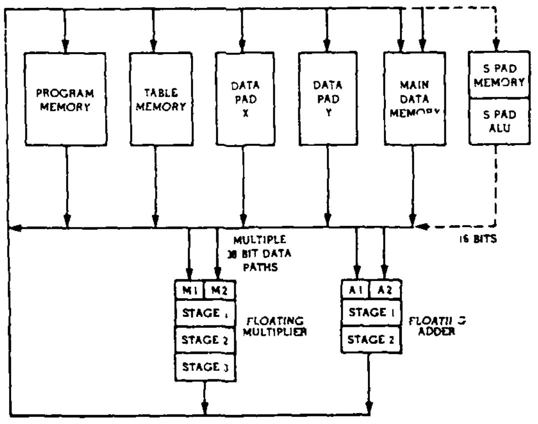


Figure 2-26
The FPS 164/264

The most significant difference in the machine is the ability to process instructions for the various units simultaneously. The instruction word is large enough to accommodate as many as 13 simultaneous instructions issued in a single cycle. While the add and multiply units are pipelined (2 and 3 segments respectively), the machine achieves impressive performance on many computational loops largely through this parallelism of functional units. For example, on sparse matrix loops which have one level of indirect addressing, the 264 can achieve upwards of 12 MFLOPS. This is, in fact, as fast as the original model of X-MPs coded optimally. In more conventional dense loops the machine has relatively short startup times when programmed in assembler. In essence the program instruction is based on the concept of algorithm pipelining as described in Section 2.1.5. The functional units can be thought of as independent processing units with special capabilities. Once a repetitive process is decomposed into a series of sequential operations of the processing units, then a pipelined series of operation can be achieved with all units operating simultaneously.

************* **IMPORTANT CONCEPTS** *************
Minisupercomputers

- o The four minisupercomputers discussed are very different in their approach to performance and architecture
- o The Alliant FX Series exploits field upgradable parallelism up to 8 CPUs, with automatic parallelization via the compiler
- o The Convex has added vector capability to more traditional minicomputer design via a cache interface. While this is similar to what IBM subsequently did with the 3090/VF, Convex has been much more aggressive in software and compiler technology.
- o The FPS-164/264 exploits a high degree of parallelism in functional units within the CPU. This is supported by the unique feature of allowing as many as 13 simultaneous assembler instructions in one cycle.
- o The SCS-40 approach is to improve on what works. By faithfully emulating the industry leader, CRAY. They have implemented an X-MP CPU with the price/performance advantages of minicomputer technology.

****

2.4 CHAPTER 2 SUMMARY

In this chapter, a survey of issues related to high performance hardware has been attempted. The ever present push for computational power seems evident in most technology based industries. No signs that this appetite for more computer power will be satiated in the near future. We have attempted to give an overview of today's supercomputers with knowledge that new designs are just around the corner. The brief descriptions and inclusion of parallel computers and minisupercomputers has been offered to help the reader be better able to assimilate new designs. The supercomputer designs of tomorrow's machines lean heavily on what has been experienced by these somewhat lesser performing machines.

Somehow the half dozen supercomputer manufacturers have evolved surprisingly similar architectures, yet the underlying chip technology and subtleties of data flow differ greatly. Whatever their similarity is today, it promises to be less in the future as the increased use of parallel CPUs takes hold. In the next chapters we will explore the software ramifications of these differences in supercomputer design.

2.5 REFERENCES FOR CHAPTER 2

[2-1] K. Wilson, "Science, Industry, and the New Japanese Challenge,"High-Speed Computation, ed. J. Kowalik, NATO ASI Series, Springer-Verlag, June 1983.

[2-2] A. Jameson, Science, Engineering and the CRAY-1, April 5-7, 1982, reported in CRAY CHANNELS Vol. 4, No. 2, 1982.

[2-3] L. Lemmerman, Aerospace Representative, Cray Research, Private Communicaton.

[2-4] R. Hockney and C. Jesshope, Parallel Computers, Adam Hilger LTD, Bristol, 1981,

[2-5] C. Ramamoorthy and H. Li, "Pipeline Architecture", Computing Surveys, Vol. 9, 1977, pp 61-102.

[2-6] M. Flynn, "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computation, Vol. C-21, pp 948-60.

[2-7] Alliant Product Summary, Alliant Computer Company, 1987.

[2-8] K. Neves , "Mathematical Libraries for Vector Computers," Computer Physics Communications, Vol. 26, 1982, pp 303-310.

[2-9] I. Y. Bucher, "The Computational Speed of Supercomputers," Proceedings of SIGMETRICS, 1983.

[2-10] J. Dongarra, J. Bunch, C. Moler, and G. Stewart, LINPACK Users' Guide, SIAM Publications, Philadelphia, 1979.

[2-11] J. Dongarra, Lecture Notes, San Diego Education Seminar, KMPS-TV, March 4, 1987.

[2-12] W. Oed and O. Lange, "On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems," IEEE Trans. Comp., C-34, 1985, pp. 949-957.

[2-13] J. Kowalik, ed, Parallel MIMD Computation: HEP Supercomputer and Its Applications, MIT Press, Cambridge, 1985.

[2-14] R. Grimes and H. Simon, "Dynamics Analysis with the Lanczos Algorithm on the SCS-40," ETA Division Technical Report, ETA-TR-43, Boeing Computer Services, Jan. 1987.

CHAPTER 3: ALGORITHM AND GENERAL SOFTWARE CONSIDERATIONS (Dr. K. W. Neves)

3.1 INTRODUCTION

It is the purpose of this chapter to delve into the relationship between hardware architecture and computational performance. The setting is scientific computation of a generic sense (i.e., application non-specific.) In subsequent chapters, the general implications discussed here will be applied to algorithms specific to computational fluid dynamics and to the major computational approaches used in that field.

In order to give a meaningful appreciation of the concepts discussed, frequent use of examples on specific hardware architectures will be utilized. Quite often several machine types will be included, and several will not. The reasons for this are two-fold: 1) It is not always possible to obtain "exactly" the same benchmark results across machine types, and 2) It is not our intent to give a definitive comparison of machine performance. These reasons may seem superficial at first glance, yet they are fundamental to the reality of the state-of-the-art of computer metrics. This is worthy of brief elaboration.

The first point relates to obtaining "exact" data. In any computation, however elemental, there are a number of hardware and software related variables that can greatly impact performance. For example, take the simple SAXPY loop analyzed in Chapter 2:

```
          J = 1
          DO 10 I = 1, N
   10     Y(I+J) = A * X(I+J) + Y(I+J).
```

Note that this is the Fortran loop for unit stride, and that quite often the SAXPY operation is employed with J equal to other integer values. The performance of this loop can depend on a host of variables, some of which are listed in Table 3-1. Rarely, are these variables listed or addressed in published hardware performance comparisons. Often the result is that the comparisons are made differently in different vendor environments (e.g., different compiler options or different systems configurations.) In order to gain access to many computers for testing, it is often difficult to assure that these types of things are treated equally. For this reason, data presented in comparing machines in this work, should be regarded as a means to assess gross comparisons of performance relative to a particular architectural feature and not a definitive study of relative hardware performance. Whenever possible, variables of the type in Table 3-1 will be indicated.

Parameters that Impact Performance

Compiler Related

- Compiler version
- Optimization level (quite often a compiler parameter)
- Rolled or unrolled (quite often performance can be improved by loop unrolling, see glossary)
- Stride
- Non-compiler optimization--i.e. assembler
- Size of loop (value of N) can affect MFLOP rating differently on different machines

Hardware Configuration Related

- Memory size and number of banks
- Cache size
- Model variances in memory access times
- Inter CPU interference in a multiple CPU machine
- Testing in a dedicated versus shared resource

TABLE 3-1

Comparing commercially available hardware is not the primary intent of this monograph. In a field such as computer performance, the most definitive work would be quickly dated by new hardware/systems releases. Thus, we have concentrated on the elaboration of important architectural features that impact performance in hopes that this information can be used in a more effective hardware evaluation methodology. In Section 3.4.5, the benchmark process, itself, will be discussed.

3.2 IMPACT OF ARCHITECTURE ON COMPUTATION

In Chapter 2, several examples were given that illustrated the impact of architectural features on performance. In particular, a generic example was given concerning the impact of varying the number of paths-to-memory on the SAXPY operation. This was viewed from an asymptotic performance view point. The computing community has long realized that asymptotic behavior analysis is often misleading. Figure 3-1 illustrates the common dilemma faced by would be purchasers of modern supercomputers. The peak performance rate quoted by the manufacturers is compared with the actual performance as provided by the Argonne Benchmark [3-1]. In each case, the performance obtained is a different percentage against the maximum. In fact, for the three machines listed the performance on the benchmark is in reverse performance order when compared to the potential peak performance, often quoted by manufacturers. There a very clear reasons for these apparently disparate facts. In fact, neither the benchmark results, nor the peak rates in the figure, are particularly accurate ratings of the three computers' relative performance!

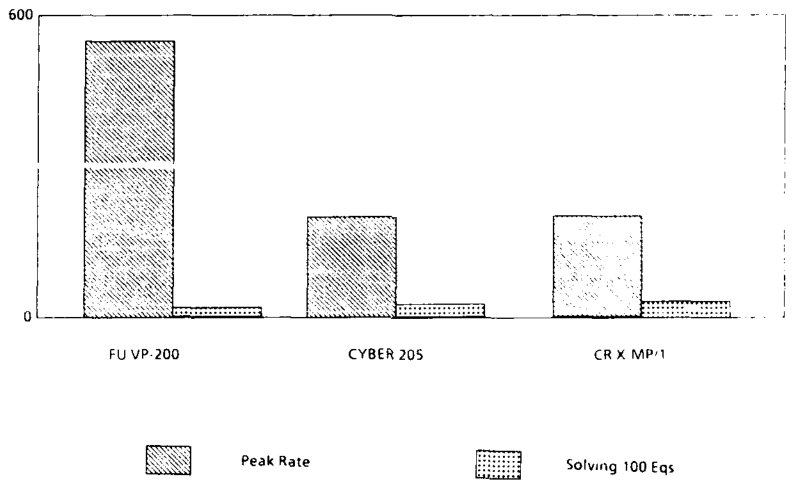


Figure 3-1
The Performance Dilemma

In the remainder of this chapter, an approach to performance analysis will be explored that not only will explain the apparent dilemma in Figure 3-1, but provides a basis for a methodology of software development for vector computing, an approach for software migration to vector computing, and a more reasonable method for benchmarking supercomputers.

3.2.1 Two-Parameter Classification of Algorithms

In Chapter 2, the parameter $N_{1/2}$ was introduced with respect to a vector operation. As a parameter, however, it can be applied to entire algorithms, or even to applications that can be parameterized by linear time dependence on N, the vector length. For example, one could compute the average $N_{1/2}$ for a matrix factorization algorithm. This could be done experimentally by randomly generating a number of matrices of size N, performing the factorization (with pivoting), and averaging the attained MFLOP rates for the family of runs. Next this could be repeated for values of N achieving a "plot" of MFLOPS versus N from which one could determine the value of N that achieves approximately half the indicated asymptotic maximum (i.e., indicated by the MFLOPS for large values of N.) While this would be a time consuming process, it certainly could be a valid comparison of machine performance (neglecting compiler and hardware configuration variances.)

In the next few sections a number of computations will be displayed using a two-parameter classification of performance, where both $N_{1/2}$ and the MFLOP rate are employed in a two-dimensional peformance space displayed in Figure 3-2. The additional insight that this simple additional parameter imparts, is rather striking, as will be observed.

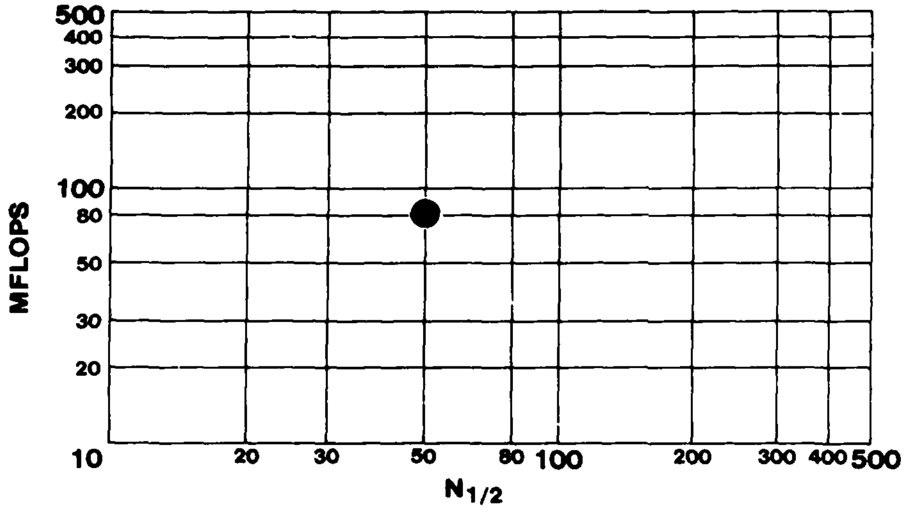


Figure 3-2
The Two-Parameter Domain

The performance point, illustrated in Figure 3-2, has the following meaning: Given an operation or algorithm that depends on "length" N, the illustrated point indicates that the performance for very large N approaches 80 MFLOPS, and for length N=50, the performance would be 40 MFLOPS. A mentioned in Chapter 2, one can observe various heuristic "rules of thumb." For example, if N is greater than 3 times $N_{1/2}$, performance is generally acceptably close to maximum. If N is less than $N_{1/2}$, performance is significantly below the rated potential of the machine.

There are a number of interesting peculiarities about the parameter $N_{1/2}$. To illustrate just one, assume that a vector operation achieves an asymptotic rate of one result (floating point operation) per machine cycle. Also assume that the operation is composed of a memory read pipe and a floating point unit pipe chained together so that the resulting combined segmentation is M segments long. The parameter $N_{1/2}$, is simply the value of N to achieve half this rate. Let CT be the cycle time, then the time to perform a vector operation of length N is given by

$$T = CT\ (M + N).$$

The time per operation is given by

$$N/T = N/[CT(M+N)],$$

or

$$N/T = \frac{1}{CT\ (\ M/N + 1\ )}\ .$$

Allowing N to approach infinity, results in the asymptotic rate in results per second, which is 1/CT, an obvious result. Half of this performance rate is 1/(2*CT). This is achieved when N=M. This somewhat surprising result stated simply is the following:

> If the operation is the result of an M segment process, and there is one result per cycle, then the $N_{1/2}$ is simply equal to the number of segments, M.

The segment length is a measure of parallelism, in that it indicates the maximum number of active operand pairs are being manipulated at one time. This term has been called the "depth of parallelism." (See [3-2].) Thus, this parameter is very related to parallelism. It is also, very clearly, an indicator of startup time. In fact in the above example, the startup time is CT*M, and since $N_{1/2}$ is, in fact, M, the parameter contributes directly to the startup time. Generalizations of $N_{1/2}$ to a purely parallel environment have been suggested by the term's originator, Hockney [3-3]. (One would expect that a generalization of $N_{1/2}$ in a parallel environment to be related to the number of processors, i.e., the depth of parallelism.)

To illustrate how the two-parameter domain can be used to display and analyze performance, the SAXPY operation and a simple vector to vector addition have been chosen as the underlying operations in order to compare several machine architectures.

Figure 3-3 displays a performance polygon for the SAXPY operation on the CRAY-1. The vertices are labeled RT, MT, RD, MD. The "T" stands for triad, indicating the operations of a scalar times a vector plus another vector (a typical SAXPY). The "D" indicates the simple dyad, or addition of two vectors. As will be observed, most machines seem to perform best on the triad operation. The "R" and "M" applies to register oriented machines. The "R" indicates the operation is being performed from data already stored in registers and the result will be placed in registers. This is, of course, an unnatural situation in that during the course of computing, the data must get to and from memory. The "M" indicates the same operations, but with data fetched and stored in memory (through registers if required).

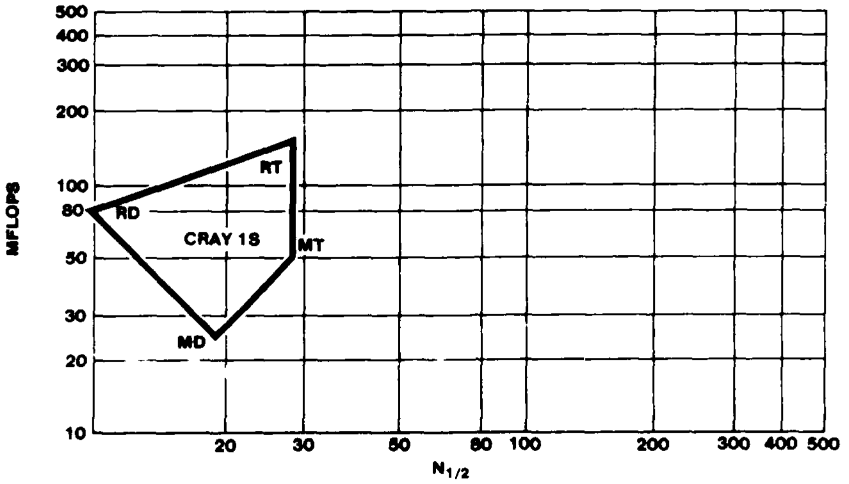


Figure 3-3
Triads and Dyads on the CRAY-1

The peak performance rate for a CRAY-1 is often quoted as 160 MFLOPS. Where does this number come from? It is possible on the CRAY-1 to put a scalar in a register then use it to multiply each component of a vector of length 64-words (sitting in a vector registers). As the vector multiply unit begins to produce results, these products can be streamed (chained or linked) as input into the add unit along with another vector from a second register. The result of this chained addition can the be stored in still another vector register. What was just described is, indeed, the triad operation whose performance point is displayed by "RT" in Figure 3-3. A simple computation, noting the 12.5 nanosecond cycle time, and the fact that two floating point operations per cycle (an add and a multiply) are produced when the pipe is full, results in the 160 MFLOP figure. The $N_{1/2}$ is approximately 28 for the RT operation; therefore, the "RT" coordinate is given by (28,160) in the two dimensional representation. The "RD" performance point indicates the register to register computation rate is (10,80) when only one dyadic operation, an add or a multiply, is being performed. This should be obvious since only one result per cycle can be produced. In the final analysis, however, the definitive performance numbers are given by "MD" and "MT", which represent the full flow of data from memory through the computational units, and back to memory. Note that the "MT" performance point, for the CRAY-1, is over a factor of 3 degraded from the potential peak rate!

Figure 3-4 is a superposition of the X-MP polygon over the CRAY-1 polygon. It is interesting to observe, in this parameter space, the result of a simple architectural change. As mentioned in Chapter 2, one of the fundamental improvements in the X-MP architecture over the CRAY-1, is the inclusion of 3-paths-to-memory rather than one. The result is a dramatic improvement in performance for the "MT" operation. As displayed for the CRAY-1, the coordinate point is (28,50). While for the X-MP, it is (13,200). Thus, on the X-MP (with the 9.5 nanosecond clock) the "MT" operation has an improved peak performance of a factor of four over the CRAY-1 (with its 12.5 nanosecond clock). The underlying chip technology accounts for only 25% of the 400% improvement. Credit the rest to an architectural improvement. Another striking feature of the Cray machines is their relatively small $N_{1/2}$ parameter values, all around 30.

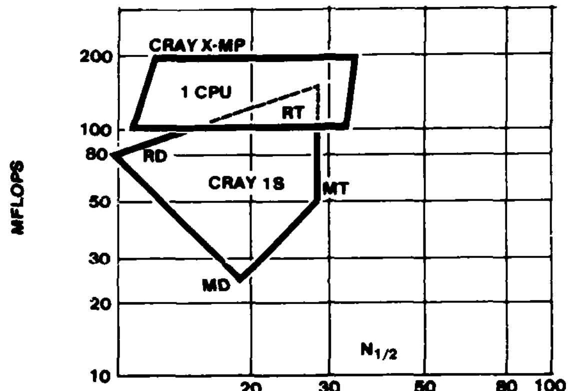


Figure 3-4
Triads and Dyads on the CRAY X-MP

Figure 3-5 adds the CYBER 205 (two-pipe machine in 64-bit mode). It is interesting to compare the CRAY-1 and the 2-pipe 205, for they competed in the market place for some time with the CRAY-1 the apparent winner. While their respective peak performance ranges were comparable, depending on stride and memory access, the CRAY-1 had a far superior $N_{1/2}$. It is generally accepted that this latter feature enabled the CRAY-1 to perform better on typical benchmark programs.

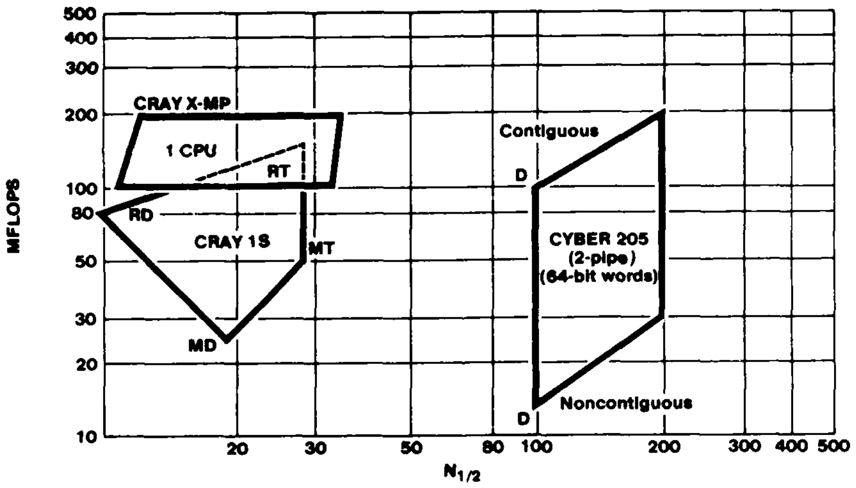


Figure 3-5
Triads and Dyads on the CYBER 205

Since the 205 is not a register oriented machine, its two-parameter polygon needs some explanation. The vertices on the left (top and bottom) represent the dyad operations and the vertices on the right (top and bottom) represent the triad operations. All operations on the 205 are memory-to-memory. The top points represent the operations when the data is accessed with unit stride, i.e., contiguous in memory. For example, if a matrix is stored by columns in memory (the CYBER 205 allows for column storage or row storage, whereas column storage is more the standard on most computers), then accessing vectors along a column will proceed contiguously. However, if one wishes to manipulate the row of a column-stored matrix, the data resides in memory with a stride of N, where N was the dimension of the matrix. On a CYBER 205, accessing with a stride of 2 results in a degradation of a factor of two in performance. Beyond two one should consider utilizing a gather operation and if necessary a scatter operation to store the result. Thus, there is a considerable degradation in performance when manipulating vectors whose components are stored with non-unit stride. This is indicated by the lower vertices in the Figure 3-5, which represent optimal access of vectors with strides greater than four using the additional gather/scatter operations provided.

It is instructive to use Figure 3-5 to make some inferences about performance on an algorithm. Consider simple matrix factorization without pivoting. The inner loop of Gaussian factorization, as it is often written, is the SAXPY operation. The algorithm can be written in column-ordered form which would access data in the inner-loop in contiguous fashion. This would enable the CYBER 205 to operate the inner-loop at maximum rates.

Referring to Figure 3-5, one can compare the CRAY X-MP performance to the CYBER 205 performance for the SAXPY inner-loop (memory-to-memory, contiguous data). The X-MP performance point for this loop is (33,200) and the 205 point is (200,200). Comparing the second coordinate (i.e., the asymptotic maximum rate for this loop), one observes that both machines are capable of achieving 200 MFLOPS in 64-bit mode. The first coordinate, $N_{1/2}$, however, differs greatly. Using the "rule of thumb" alluded to earlier, one can now estimate an upper bound on performance for a specific problem size. Gaussian elimination can proceed at no faster rate than the inner-loop. Let's assume a problem size of 200. At each stage of Gaussian elimination the loop length will shrink by one; thus, the maximum computation rate in MFLOPS will be dominated by the performance at loop length of 200. For the CYBER 205, $N_{1/2} = 200$. Therefore, the loop performance at N = 200 is 100 MFLOPS. The $N_{1/2}$ for the X-MP is only 33; therefore, N = 200 is well over three times the $N_{1/2}$. Consequently, the maximum loop computation rate will be very close to 200 MFLOPS. As the size of the inner-loop shrinks the performance edge favors the CRAY even more. Using this line of reasoning, it is not difficult to see why these two computers, rated at the same peak performance rate, perform differently on this algorithm and problem size. In fact, referring to Figure 3-1, this is a partial explanation for the disparity between the 205 and X-MP on the benchmark performance, which was for problem size 100.

Some insight into the impact of parallelism can also be gained from examining these two-parameter diagrams in a multiple CPU or multiple functional unit environment. Figure 3-6 can be used to see the effects of two different approaches to replicative parallelism. In the case of the CRAY X-MP, parallelism is introduced by adding another CPU. Whereas, in the case of the CYBER 205, optional pipeline units are added to the architecture. The effect on a triad operation is the same for each machine, the $N_{1/2}$ and the peak rates both double. In fact, if the goal of introducing parallelism is simply to improve performance on SIMD processing of elementary loop calculations, the approach of adding functional units is far simpler, and just as effective as adding CPUs. However, by adding CPUs the possibility of MIMD processing is created. There is no doubt that the multiple CPU approach will be preferred in future machines because of this flexibility and because of the limits to which we can produce highly vectorizable programs with very long vector operands. In fact, since the $N_{1/2}$ generally doubles with a doubling of the depth of parallelism, the vector length for efficient operation doubles as well.

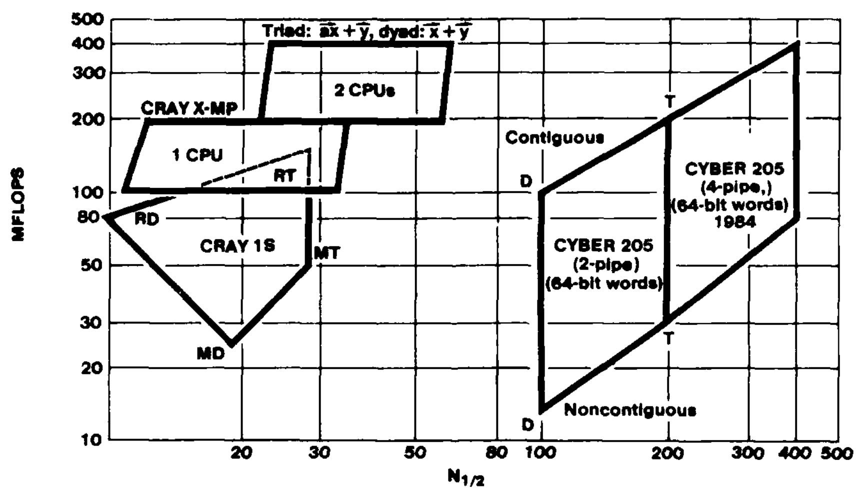


Figure 3-6
Parallelism in Two-Parameters

Figure 3-7 adds the Fujitsu VP-200, however this polygon is slightly different. The four corners are all memory to memory operations. The left-top vertex is a MT. The left-bottom vertex is a MD. The right hand vertices (top and bottom) are the loops without unit stride. The VP-200 loses one path to memory in the processing of strided loops, causing this degradation. An awareness of this degradation in algorithm design for the VP-200, can mean a factor of two or more in performance.

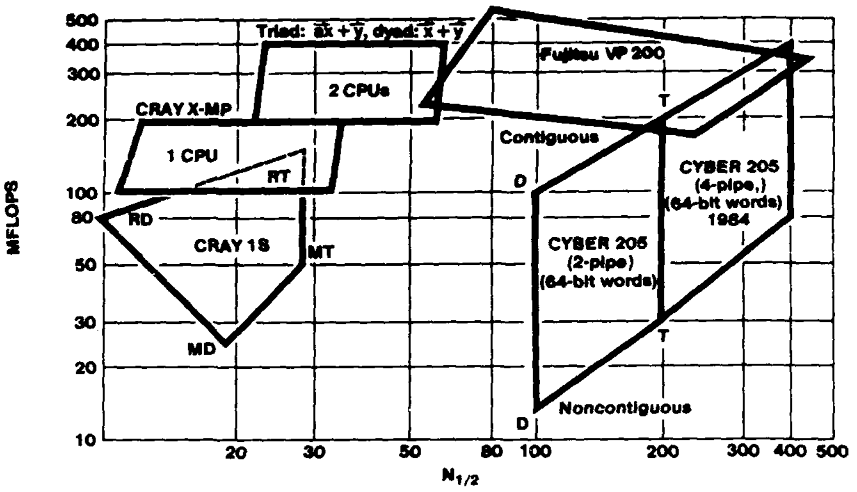


Figure 3-7
Triads and Dyads on Several Supercomputers

The dense vector dyad and triad operations data are the ideal operations for most vector computers. These machines were literally designed for high performance on these operations. Unfortunately, all of science does not rest on these two operations. To rate computer performance solely on the performance of these loops is little better than basing one's assessment only on the peak MFLOP rate. To illustrate this, consider a slightly more complex computation built upon dyad and triad operations, the matrix factorization. With an awareness of the underlying architecture and its strengths and weaknesses, one can minimize the impact of obvious computational bottlenecks through clever instruction selection or algorithm alternatives. Sometimes compilers can accomplish improvement through clever scheduling algorithms, but the substantial gains are made at the algorithm level. A well known technique to avoid to path deficiencies can be utilized in matrix factorization. The concept was developed for the CRAY-1 which often had performance problems on key loops because of the single path to memory. Fong and Jordan [3-4] coined the phrase "supervector" performance for the technique. A typical Gaussian elimination algorithm kernel based on the SAXPY is given below.

```
      DO 10 I = 1, M-1
         DO 10 K = I+1, M
            A(I,K) = A(I,K)/A(I,I)
            DO 10 J = I+1, M
               A(J,K) = A(J,K) - A(J,I)*A(I,K)
 10   CONTINUE
```

The order of these loops can be changed to produced a column-ordered elimination as follows:

```
      DO 10 K = 2, M
         DO 10 I = 1, K-1
            A(I,K) = A(I,K)/A(I,I)
            DO 10 J = I+1, M
               A(J,K) = A(J,K) - A(J,I)*A(I,K)
 10   CONTINUE
```

The inner-most loop in either formulation is the SAXPY. At face value, it has two vector memory fetches and one vector memory store. However, with a bit of reflection, one can observe that in the second formulation, the vector A(J,K) for J = I+1 to M, is being used as an accumulation array and can be saved (if there are registers) for the next second level loop iteration on I. This can be done in slices equal to the vector register length and can proceed with only one vector memory fetch, for A(J,I); J = 1,M. The final store of A(J,K) occurs only at the outer-most loop level (for each vector slice) when K is incremented. This is a dramatic decrease in memory traffic, and, on a path limited machine, it is possible to improve performance as significantly as adding needed paths!

One can look at a few generic architectural parameters and observe performance behavior using "proper" algorithm techniques. In Table 3-2, four machine characteristics are compared in an "order-arithmetic" fashion. [By order arithmetic, we refer to a common terminology often used in complexity and numerical analyses wherein only powers of salient terms are used and minor additive constants are ignored. For example, in counting the number of cycles of a vector operation of length M, one would use O(M) to indicate that for large M, the startup times are negligible. If multiplicative terms are consequential, the can be included. The interpretation of such notation is usually clear from the context, and will be used occasionally in this chapter.] The first 3 variations are using 1, 2 and 3 paths, and the fourth variation is a 3-path, but non-register machine. This latter machine could be likened to a CYBER 205. The CRAY-1 is a one-path register machine, the Fujitsu VP-200 and the FPS 264 are two-path register machines; and the CRAY X-MP is a 3-path register machine. The table, however, is a study in a generic setting, the fact the characteristics exist in the commercially available architectures mentioned is an aside. Loop 1 is the SAXPY and Loop 2 contains four vector memory references (3 fetches and 1 store).

Achieving Efficiency

```
LOOP 1: A * X(I) + Y(I) ;  I = 1, N
LOOP 2: A(I) * X(I) + Y(I) ;  I = 1, N
```

TIME IN CYCLES
(neglecting startup)

	LOOP 1		LOOP 2	
	VECTOR	SUPERVECTOR	VECTOR	SUPERVECTOR
1 PATH/REG.	3N	N	4N	2N
2 PATH/REG.	1.5N	N	2N	N
3 PATH/REG.	N	N	1.5N	N
3 PATH NO REG.	N	N	2N	2N

TABLE 3-2

The table illustrates several conclusions worth mentioning. First, a non-register machine cannot ameliorate path deficiencies through tricks like supervector programming, for this technique is register oriented. Supervector programming can produce dramatic performance improvement as noted by Loop 1 on the 1-path machine. In fact, this technique is used in most CRAY-1 algorithms for dense matrix factorization. When the number of paths is less than the number of memory references the "trick" described in Figure 2-20 in Chapter 2 can be employed to achieve worthwhile improvements as indicated by both the 1.5N entries in the table. Finally, on Loop 2 one observes that supervector programming with registers, even on a 1-path machine, is as effective as 3-paths-to-memory on a non-register architecture. This analysis has ignored other significant factors such as startup time, but serves as our first in-depth illustration of how architectural features can interplay with algorithmic strategies to produce "better" than expected results.

*************** IMPORTANT CONCEPTS ***************

o Performance is affected by a curious mixture of compiler, operating systems, hardware architecture, and algorithm selection.

o The peak performance possible (usually rated in MFLOPS) is a function of hardware only. Whereas, achieving the highest possible performance on a specific computation is a software issue.

o $N_{1/2}$ (the length of a vector to achieve half the peak performance) is another useful parameter for rating performance.

  - If vector length is consistently less that $N_{1/2}$, vector operations will be poor in performance.

  - If vector length is greater than three times $N_{1/2}$, performance in vector mode will be close to maximum.

  - $N_{1/2}$ is a way to measure the depth of parallelism inherent in a process on a vector machine, and has a natural analog on parallel machines, the related to the number of CPUs.

o The features discussed in Chapter 2, paths-to-memory, vector storage restrictions, registers if well understood can be utilized in unique ways to improve the computational performance of simple algorithms.

****

3.2.2 Case Study: Sparse Matrix Computation

Matrix factorization is a fundamental algorithmic tool of scientific computation. Recognition of this fact gave birth to vector computing, for it was realized that most dense matrix manipulations and solution processes deal with vectors. However, while this hardware revolution was taking place, algorithm researchers were busy conquering another aspect of computational limitations, that of lack of memory. It is very common in scientific computation to deal with matrices that are not dense (i.e., full of zeros). In partial differential equations, structural analysis, network related computations, and a host of other areas, the naturally occurring matrices are sparse. In some applications the occurrence of the non-zeros is very structured. Other times the occurrence of the non-zeros is random. Motivated by storage economy and dramatic reduction in the number of computations to solve a sparse equation, researchers have developed a number of robust algorithms that create a tremendous savings of computer time. This, in time, allows for the solution of significantly larger problem classes, irrespective of computer power. In such algorithms the complexity of the solution process is increased, however, by the introduction of storage schemes for the sparse vectors. Hardware manufacturers took some time to respond to these algorithm advances by providing sparse vector instructions implemented in hardware.

Historically, the CDC Star-100 computer attempted to accommodate the "sparse vector" construct in both hardware and software. The Cray series of supercomputers largely ignored these algorithmic kernels in their instruction set. In 1983, Fujitsu and Hitachi introduced their supercomputers which included hardware instructions for "gather/scatter" operations that most sparse matrix computations require. In addition these Japanese manufacturers took a very aggressive approach in compiler optimization that utilized these instructions to improve the performance of otherwise non-vectorizable loops. In apparent response to the popularity of this instruction, Cray Research introduced an upgrade to the X-MP line in 1983 that includes the gather and scatter instructions. Since then new computers such as the NEC SX-2, the CRAY-2, and the 3090/VF support this instruction type. In this section, a closer look at this instruction class will be used to reveal the impact of hardware changes of this type on computation.

Figure 3-8 lists and illustrates a basic Fortran loop employed in direct elimination methods for factorization of sparse matrices. Not all sparse techniques use this particular loop. Some algorithms exploit particular "structure" or sparsity patterns. This loop, however, does represent both the computation and the data structures that have been used most often. Some of the sparsest matrix problems occur in electric power distribution systems. In this class of application, it is not unusual to have the sparse loop in Figure 3-8 accounting for 80% of the CPU time.

```
       DO 10 I = 1, N
          J    = Index(I)
10        Y(J) = a * X(I) + Y(J)
```

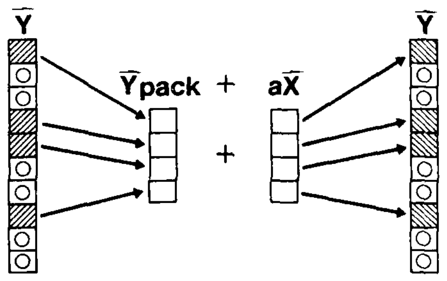


Figure 3-8
The Sparse Matrix Loop

As illustrated the loop relies on an "index" vector to keep track of the location of the non-zeros in the "full" vector (typically a column in a matrix). The full mathematical vector is rarely stored. Its non-zeros are stored and expanded to full vector form only in the sense of the above loop. With these methods the number of operations for common sparse matrix operations is "order-M", O(M), as opposed to O(M**3), where M is the matrix size. A detailed description of this class of factorization algorithm is given in Dembart and Neves [3-5]. The challenges in sparse matrix algorithm design actually begin in development of effective ordering schemes that will reduce the amount of "fill-in" (creation of new non-zeros) that occurs during the numerical solution. A description of ordering algorithms is beyond the scope of this discussion. The reader is referred to a more general discussion of a variety of sparse algorithms and ordering schemes in Duff, Erisman and Reid [3-6]. The timing of sparse instructions are the crux of their effectiveness. Startup time (reflected in $N_{1/2}$) and the asymptotic rate are both critical. However, the range of loop sizes does not tend to get enormously large. For this reason, in this section the entire performance curve over the vector length will be examined whenever possible.

The CDC Star 100, CYBER 203 and the CYBER 205 implemented the same set of hardware instructions. Their instruction set suitable for the vector loop above, was quite rich with alternatives. In fact, today's supercomputers offer similar choices. Early versions of CDC's vector compilers even had sparse vector constructs added to Fortran. These were eventually dropped for a number of reasons. The types of control mechanism employed by these machines fell into two categories. One was the index vector approach, and the other was the control vector approach. The latter employs a bit-vector. A "1" in a bit location signified a non-zero. (At least this was a typical usage.) The bit control vectors could be used to suppress storage of a result, but not its calculation. It is possible to construct seven different methods for processing the Figure 3-8 loop. One must realize throughout this discussion that there are two basic loop parameters. One is M, the length of a full vector (the dimension of the matrix being solved.) The other is N, the number of non-zeros in a given row. The tacit assumption is that, for algorithms of this class, M >> N. Table 3-3 below lists the different methods.

Sparse Loop Complexity on CYBER 200 Series

	METHODS	ORDER ARITHMETIC
1.	SCALAR FORTRAN	O(N)
2.	VECTOR FORTRAN USING FULL VECTORS-IGNORING SPARSITY	O(M)
3.	COMPRESS/EXPAND (LIKE GATHER/SCATTER ONLY USING BIT-CONTROL)	O(M) + O(N)
4.	THE SAME AS 3 WITH PROVIDED LIBRARY ROUTINES	O(M) + O(N)
5.	EXPAND BY BIT, USE FULL VECTOR OPS VIA BIT CONTROL	O(M) + O(N)
6.	ADD NORMAL SPARSE (FORTRAN PROVIDED CONSTRUCT)	O(M) + O(N)
7.	TRUE GATHER/SCATTER VIA INDEX	O(N)

TABLE 3-3

As far as asymptotic performance, one can quickly infer form the table that loops 1 and 7 should prevail as the sparsity increases. Any method with O(M) manipulations will be slower than an O(N) method as M approaches infinity, assuming N is bounded. The question as to actual performance when M is finite and N varies, is a different matter.

(Note: the CYBER series computers indeed have the gather/scatter constructs. One very significant difference between these machines and the other supercomputers is the memory-to-memory limitation. A memory-to-memory gather operation of a disperse vector into a contiguously packed vector is not chained or linked into a functional unit. It is a preliminary step which is followed by a vector operation. The CRAY X-MP, CRAY-2, Fujitsu VPs, Hitachi S-810, NEC SX, and the IBM 3090-VF, the random placement of data can be streamed into functional units without this intermediate step. Note that the IBM 3090 VF has the functional units cache interfaced which can degrade performance due to frequent cache hits resulting from widely scattered data.)

In fact, the subtleties of architectural balance are very apparent in this discussion. The relative speed of scalar and vector operations become a fundamental performance issue with respect to algorithm selection. From an architectural balance point of view, the three CDC machines can be characterized as follows:

STAR-100 - slow scalar performance relative to vector speed

CYBER 203- fast scalar performance same vector speed as 100

CYBER 205- 203 scalar speed with very fast vector speed

Figures 3-9 through 3-12 below display the fact that architectural balance can greatly impact algorithm selection. The figures display the best performing algorithm over the two-parameter plain of M and N. Recall that N is less than M, and that for most interesting applications of the sparse loop, N is very much less than M. The diagonal line represents N=M, the case when the number of non-zeros equals the full vector length (i.e. dense).

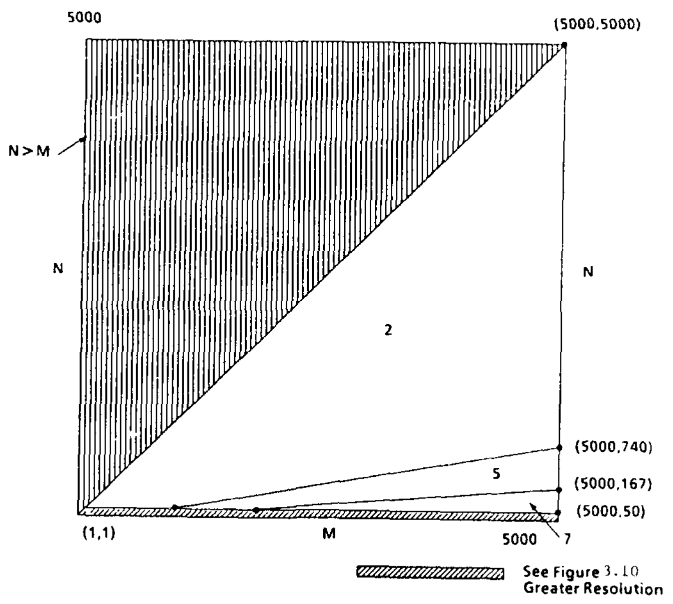


Figure 3-9
Density Performance Diagram - Star 100, part 1

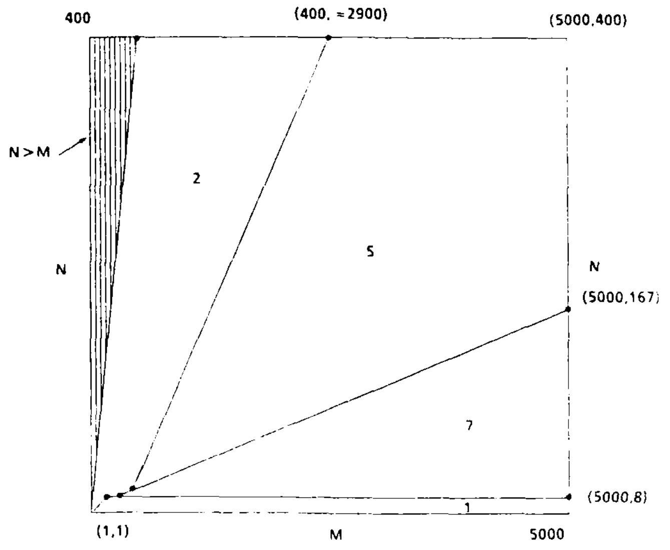


Figure 3-10
Density Performance Diagram - Star 100, part 2

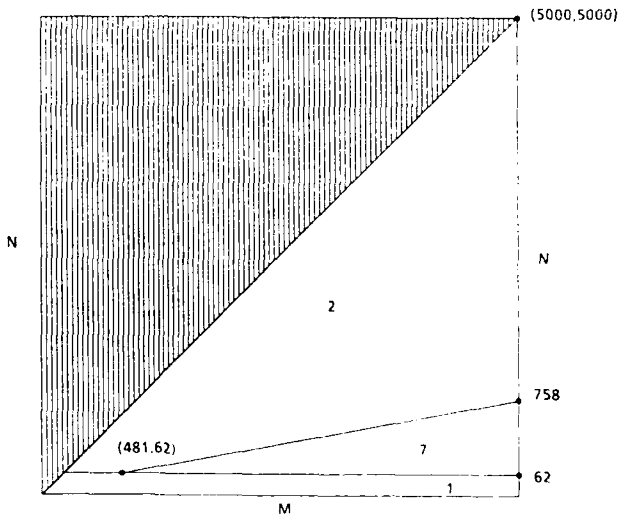


Figure 3-11
Density Performance Diagram - CYBER 203

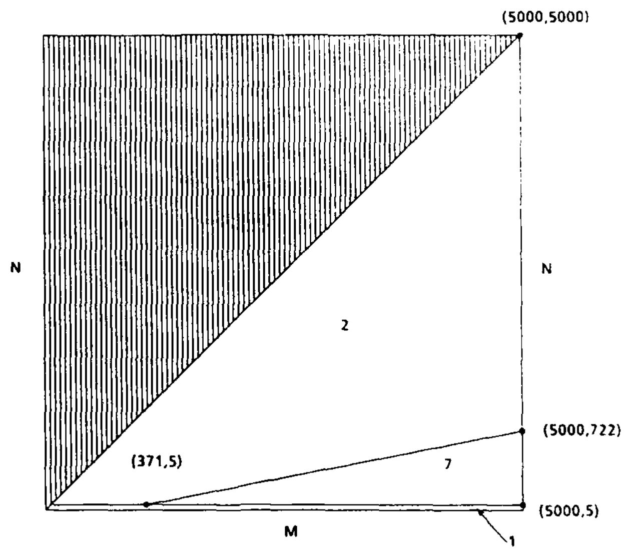


Figure 3-12
Density Performance Diagram - CYBER 205 (2-pipe)

At 10% density (or less), sparse algorithms can be quite advantageous. Methods 1, 5, and 7 are the methods of choice in the less than 10% sparse range, with method 5 being effective only on the CYBER 203. In fact, since the size of the inner loop and density actually vary during the course of a factorization, a truly optimal method might be a hybrid method, if low overhead switches could be worked out. It wasn't until the CYBER 205, however, that the hardware gather/scatter approach was successful enough to dominate the process over the most interesting ranges for N/M. Experience reveals that the common upper value for the number of non-zeros in a column, with a good ordering algorithm can remain surprisingly small. This is of course application dependent. In the most sparse problems that we have encountered, electric power transmission problems, it is not unusual to have matrices of order 5000 with less than 40 non-zeros in any column after fill-in. In Navier-Stokes analysis, the matrices can be quite sparse as well, although somewhat more structured. With the advent of gather/scatter operations in hardware, new approaches to matrix solutions in this application area should be explored. In other fields, it has been demonstrated that ignoring the structure of the matrices in favor of random sparse methods, can lead to performance gains.

The limitation of the CYBER 200s vector floating point operations to contiguously stored vectors, required the eventual improvement of the gather/scatter operations, for they are also used in dense manipulations on the CYBER series in order to effectively manipulate strided vectors (such as the row of a column-stored matrix). Nevertheless, the utility of these instructions in sparse manipulations proved important in so many applications, that they have become an important part of the (de facto) standard set of vector operations in most of today's supercomputers.

The detail offered here on the various methods (for performing this inner loop on the CYBER series) is illustrative of similar choices required on the Japanese machines which employ an optimizing compiler for the removal or optimization of "IF-tests" in DO-Loops. This will be discussed further in Section 3.4.2.

A case study of the sparse loop would not be complete without examining its implementation on several different machines. In addition, it will be valuable to examine the impact of how variances in the loop's performance impact the entire factorization algorithm. Figure 3-13 displays the performance of the sparse loop on a computer, the CRAY-1, without the hardware instructions that support vector gather/scatter. Three MFLOP-curves are given plotted against vector length. One can determine the $N_{1/2}$ values for each curve by dividing the peak performance by 2 and finding the value for N at which the curve crosses its half maximum value. The Fortran curve is the result of the CFT 1.13 compiler, and is the result of scalar processing. The curve labeled GSS is a combination of an assembler version of a gather routine, a

SAXPY routine, and a scatter routine, all written in nearly optimal fashion. This three module approach would mimic the hardware approach required on a CYBER 205 with a vector gather, vector triad, and vector scatter operation. The top curve is simply one module coded with as much overlap and optimization as is possible. All three curves are, of course, scalar subroutines since the CRAY-1 has no vector hardware features to employ for these loops.

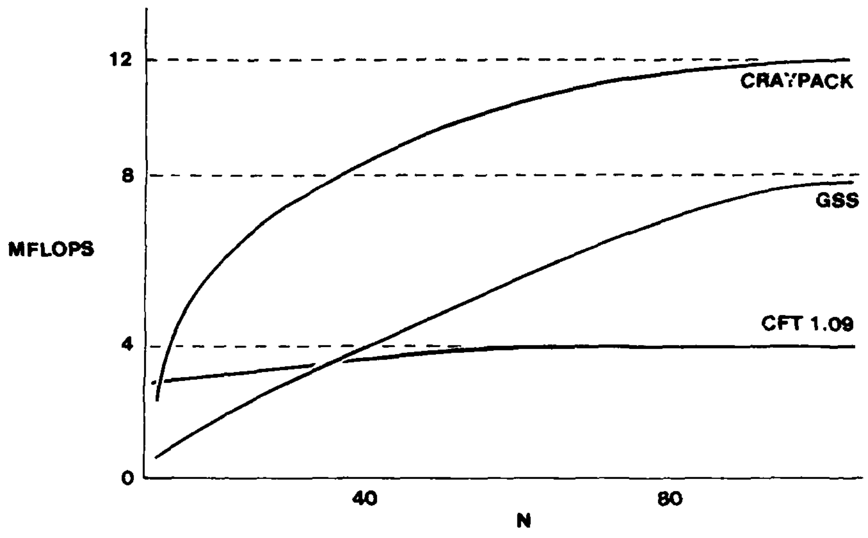


Figure 3-13
Sparse Loop in Software - CRAY-1

One can observe that the asymptotic maximums are quite different. The fastest loop is three times more powerful asymptotically than Fortran. The GSS loop is only twice as fast. Table 3-4 gives the factorization times on two matrices, one happens to be from a very sparse power systems problem, and the other from a large structural analysis problem.

Sparse Factorization Times - CRAY-1

	STRUCTURES PROBLEM	POWER PROBLEM
FORTRAN	1.2 sec	0.181 sec
GSS	0.9 sec	0.240 sec
SAXPY	0.6 sec	0.168 sec

TABLE 3-4

The structures example conforms to one's intuition. The factorization time for the algorithm using the Fortran loop is the slowest and the others show improvement in a fashion that corresponds (somewhat less than linearly) to their relative loop maximum rates. Since the inner-loop on this problem does not totally dominate the computation, the improvements are slightly less than linear. The performance of the power problem's matrix factorization, however, shows some anomalous behavior. The SAXPY is only slightly faster than Fortran, and GSS as an inner-loop, in fact, significantly slows the performance. To see why these results are appropriate, return to Figure 3-13 an observe that the $N_{1/2}$ for the GSS curve is very different. The $N_{1/2}$ values are roughly as follows:

	$N_{1/2}$
FORTRAN	20
GSS	60
ASSEMBLER	20

As a result, the performance for small values of N can be very different. In fact, for N < 35, the GSS loop is slower than the Fortran loop. It so happens that the structures problem often has well over 40 non-zeros per row, consequently the loops are exercised at values of N large enough to behave more like their respective maximum rates. The power problem, however, always exercises the loops with values of N < 40, causing the apparent anomaly. This example, although not related to vector programming, illustrates a basic concept: kernel computations can only be used to predict performance if there is a thorough knowledge of how application programs exercise these kernels (i.e., knowledge of what parameters the loop will be given throughout the computation.)

Figure 3-14 displays the loop performance curves for the other machines listed. The loops are written in near optimal assembler for each machine listed, as opposed to Fortran. The dotted line for the CYBER 205, is method 7, displayed in Figure 3-12. Note that even though this is now a "vector" operation, the cross-over point for the CRAY-1 in assembler is surprisingly large at N=45. In fact, the power problem would be faster on the CRAY-1, a totally scalar approach, but with superior scalar speed. The more modern implementations of the gather/scatter operations, lead to the more impressive timing curves for the X-MP and the Fujitsu VP-200. (Timings for the Fujitsu VP-400 and the NEC SX-2 were not available, but could prove interesting--particularly, the NEC SX-2 with its relatively fast clock.) Since the X-MP curve has a lower asymptotic maximum (93 MFLOPS) than the Fujitsu VP-200 (at 176 MFLOPS), the curves will cross (not displayed). The cross-over point is about N = 100. Applications that lead to denser, yet randomly sparse, matrices could conceivably operate with over one hundred non-zeros in a single row with fill-in. In fact, as problem sizes become larger this is likely. For those problems the Fujitsu machine will improve relatively to the X-MP, at least at the inner-loop level.

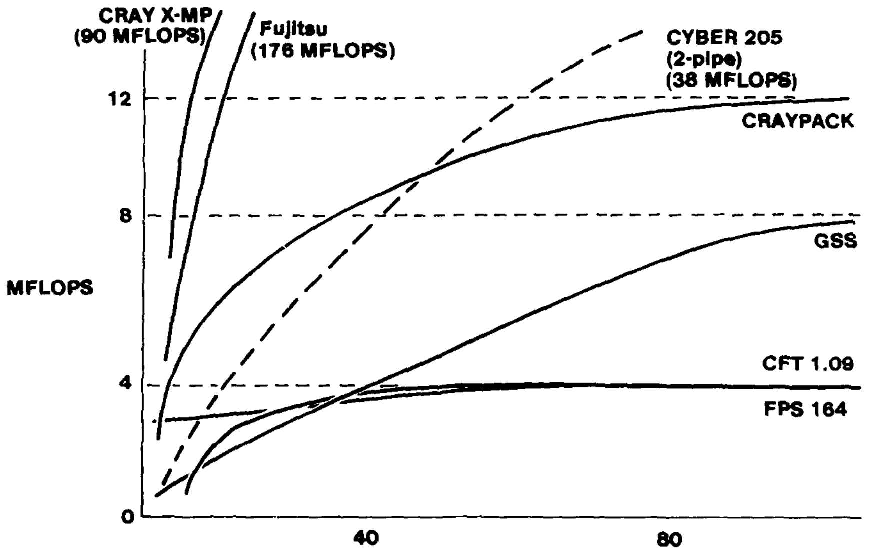


Figure 3-14
Sparse Loop in Vector Mode - CYBER, Fujitsu, X-MP

Table 3-5 illustrates the improvement possible when a function such as the sparse loop is supported by vector instructions in the proper manner. The X-MP timings are single CPU times on an X-MP that supports vector gather/scatter (9.5 nanosecond clock). The problems in the table are actual sparse matrices that arise in oil reservoir simulations for a large US oil company. The performance improvement (factor 2.9 to 3.4) is rather surprising considering that only 1.25 is due to cycle time improvements, and the balance due to one small Fortran loop.

Sparse Factorization Times: CRAY-1 vs. X-MP

MATRIX DIMENSION	CRAY-1S	CRAY X-MP	SPEEDUP FACTOR
1080	2.559 sec	0.745 sec	3.4
5005	2.942 sec	0.959 sec	3.1
1104	0.141 sec	0.070 sec	2.0
3312	2.032 sec	0.701 sec	2.9

TABLE 3-5

************* **IMPORTANT CONCEPTS** *************

It may seem that an inordinate amount of discussion has taken place on one simple loop. The loop arises in many fields and is an important loop. However, the concepts discussed in this case study are indicative of basic performance issues in vector computing.

- o Quite often important algorithms or even entire applications, have key computational kernels that account for a substantial percentage of computing time.
- o Time consuming computational kernels can be extremely sensitive to architectural features.
- o Converting scalar operations to vector operations doesn't always mean performance will improve.
- o For some computational kernels $N_{1/2}$ can be as important as the asymptotic performance rate.

****

3.3 PRINCIPLES OF VECTORIZATION

Any exposition on vectorization should logically begin with a law attributed to Gene Amdahl, although it was independently stated by several persons. The application of the law to vectorization is discussed here including the ramification of parameters such as $N_{1/2}$. (A statement of Amdahl's for parallel machines is given in the glossary.)

The law itself is a statement of the intuitively obvious: If one has both, a high performance computing engine and a lower performance computing engine, then executing a greater percentage of your application on the high performance engine will result in greater execution speed. If the high speed engine is a "vector" functional unit and the lower speed engine is a "scalar" unit, then performance is improved by "transferring" more of the computational process to the vector unit. This ""transferring process" is often termed "vectorization". To make these heuristic comments more precise, let V be the vector speed of a process and S be its scalar speed. Call G the gain in speed by performing the percentage P of the process on the vector units. The performance gain, G, is given by Equation 3-1 below:

$$G = [(1 - P) + P/R]^{-1}, \qquad (3\text{-}1)$$

where R is the ratio of the improved vector to scalar spe ds, V/S. (Note that P is expressed as a fraction between zero and one in the formula.) Figure 3-15 displays the characteristic curve for ratios of R = 10 and R = infinity. The former ratio is typical of current vector/scalar technology. The latter ratio, represented by the dotted line, is the impossible case of an infinitely fast vector unit. The horizontal axis represents the percentage of the computation time moved to the vector (faster) computational engine. One must be very careful to observe that this percentage should be interpreted as the striking fact that 100% "vectorization" with R = 10 is equally as effective as the impossible situation of 90% vectorization on an infinitely fast vector unit.

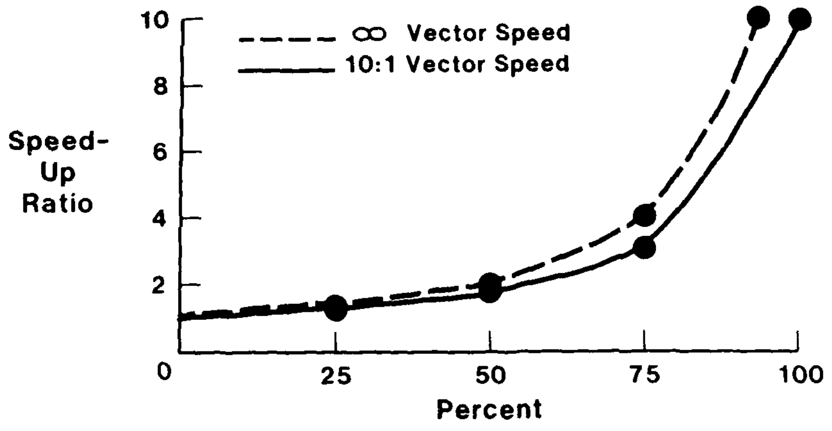


Figure 3-15
Amdahl's Law

Figure 3-16 gives another look at the same phenomenon, at 75% vectorization. At this percentage of vectorization, the ability to infinitely increase the vector unit's performance would improve overall performance from 3.63 times scalar speed to 4 times scalar speed. (This is indicated by the arrow upward to the dotted line.) However, the striking result is that, by increasing the percentage of vectorization to 90% from 75%, the performance goes from 3.63 times scalar to 5.26 scalar speed. This latter improvement is entirely a software design issue. Nevertheless, the pay-off is greater than that attainable by the impossible hardware improvement suggested. This illustrates that architectural improvements are really only opportunities for performance improvements, and that the opportunity is not realized except in software that can take advantage of the hardware design.

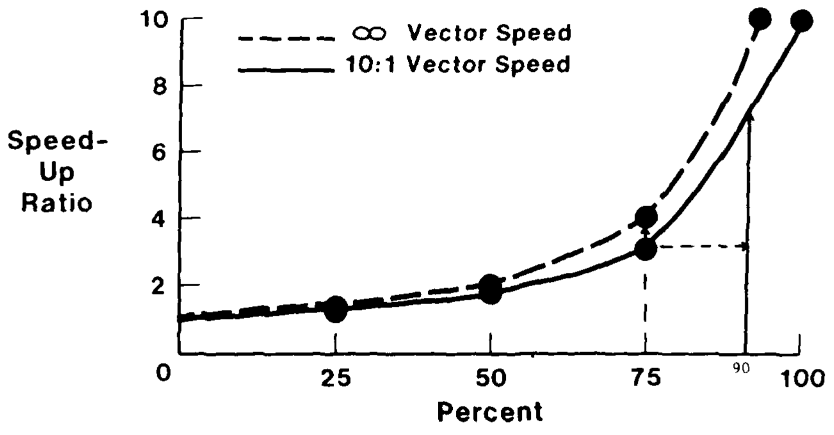


Figure 3-16
Amdahl's Law: Implications

Unfortunately, Amdahl's law as stated above is, in some sense, only an ideal case. When one assumes the vector unit speed is, say, ten times the scalar speed, the implicit assumption is that the asymptotic performance is ten times faster than the scalar speed.

As pointed out in Chapter 2 (see Figure 2-13), the performance for short vectors can be very different. In fact, recall that $N_{1/2}$ was defined to be the vector length to achieve one half the full performance. Other performance related vector lengths could be defined. For example, one could define N_b as the "break even" vector length. That is, the length of a vector for which scalar performance is equal to vector performance. When one uses segmented pipelined approach to performance improvement, the penalty is that processing only a single operand pair takes longer than doing it in scalar mode. Consequently, there is a length for which the two approaches are roughly equal -- N_b. Figure 3-17 displays the performance ranges for Amdahl's Law with various vector lengths (N = infinity, $N = N_{1/2}$, $N = N_b$). The fact is that this chart is quite realistic. For example, the hashed region below the N_b curve is for vector lengths less than the break even vector length, meaning that scalar performance is better than vector performance for these vector lengths. As is indicated in this region -- the greater the vectorization, the poorer the performance! Indeed, it has been observed in actual application programs that performance can degrade with compiler vector optimization options turned on, versus left off. In fact, closer examination of the programs of this type often reveals that for the algorithm being used the most, the N_b is quite large, and the average vector length used is small. With some work this situation can often be rectified.

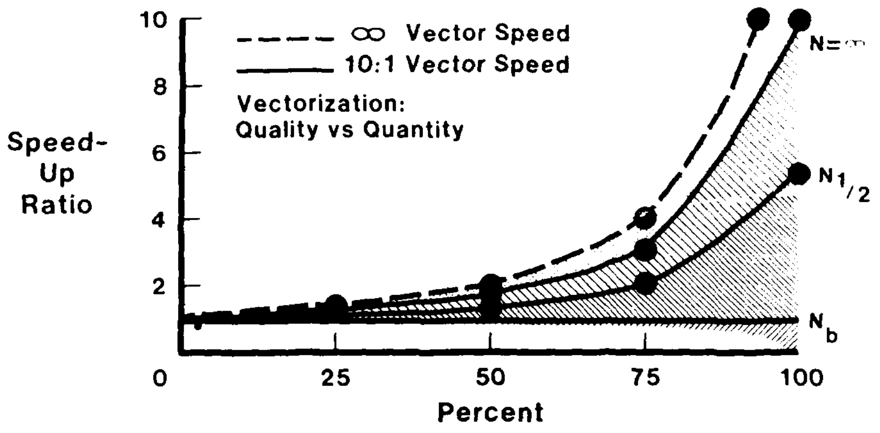


Figure 3-17
Amdahl's Law: The Reality

The implications of Amdahl's law are clear. The introduction of pipelined/parallel function units to improve performance is not a panacea. The necessity of attaining a high percentage of vectorization is fundamental, but it is equally important to have "high" quality vectorization--i.e. operations with long (very much greater than $N_{1/2}$) vectors. Quality also in the sense of appropriate usage of machine characteristics. For example, we observed previously (in Figure 3-11) that for at least one operation, the vector mode was slower than the scalar mode for certain sparse operations. One must not forget that sometimes vectorization alone is not necessarily the best approach to code optimization.

************ IMPORTANT CONCEPTS ************

- Amdahl's law indicates that the speed of vector processors is significant provided software can be effectively "vectorized", i.e. a high percentage of the computation time can be executed on the vector "side" of the machine.

- A high percentage of vectorization is a necessary, but not sufficient condition for high performance. The vector operations performed must be executed at the high performance range of vector length. A rule of thumb: a vector is "long" if it is three times the length of $N_{1/2}$.

- By their design, pipelined functional units are generally slower than scalar units when performing one single operation. Therefore, there is a vector length, N_b, for which vectors of smaller length are slower than scalar processing of the same operations.

****

3.4 SOFTWARE MIGRATION ISSUES

Although computing and programming have a relatively short history, the world's technology has become very dependent on computation. Advances in science and engineering are increasingly dependent on digital computation. Many industries rely on "production" programs. These are programs that are characterized by repeated usage that provides fundamental information for manufacturing or process control vital to the economic well-being of the company involved. The vast majority of computers sold in private industry and governmental laboratories are used to satisfy the computational requirements of these programs. In aerospace these programs include design tools such as CFD analysis programs, structural analysis, flutter, guidance and control, electromagnetic analysis, and host of scientific tools required by the engineering community. The monetary value of this software in almost any terms one uses, far out weighs the cost of computer hardware. The person-years of labor devoted to the care and feeding of an important production program can be enormous. Consequently, it is not surprising to discover that many of these programs will survive several computer systems, dominate computer loads, and run on a variety of equipment within a single company.

The issues related to software migration are among the most costly in the computing process. The conversion of programs from one hardware environment to another (even of similar underlying architecture) can consume many labor resources. New computer designs, such as parallel machines, will probably will not be truly successful until a sound software migration strategy is found for the enormous world-wide investment in production programs. The technology of the computer industry must be brought to bear on this migration problem. It is not simply an internal architecture problem, compiler problem, or algorithm challenge. Disk and I/O performance and system throughput are fundamental in large-scale computing performance. In this section we will explore software migration issues related to vector computing, and briefly describe some of the challenges of the parallel future that has been predicted.

3.4.1 Transportability: Is it Feasible?

The concept of "transportability" is itself a compromise, or recognition of a, perhaps, impossible goal. In order to discuss transportability one must appreciate the goal. The goal is portable software. For applications related to scientific computation the term has grown to mean the ability to run (compile and execute) a single source Fortran program on a variety of computers. In light of the large differences in today's computer architectures, one would think this concept is almost beside the point. Yet, the desire to have programs run in a multi-vendor environment is strongly motivated by increased use of distributed, networked, and/or multi-tiered computing systems. With the apparent growth of computational power within the single workstation, and the variety of mini-, supermini-, and minisuper-computers, it is logical that entire programs or parts of entire programs will be required to be functional in several environments. Transportability is a rather loose term, which we will take to mean "as portable as is practical."

The issue of true portability is not only a difficult technical challenge, but a critical issue in reducing long term software development and maintenance costs. The most active area of application of portability over the years has been in mathematical subroutine libraries. A brief historical perspective is in order.

The developers of reusable mathematical software comprise a rather small community, yet their products are used by almost all modern computational scientists. In providing reusable tools for a broad set of users, over a broad collection of computers, these developers began to deal with portability issues in earnest in the mid-1960's. The biggest issue was to achieve portability without sacrificing efficiency. (For an overview of the issues faced mathematical and statistical library developers, see [3-7], [3-7], and [3-8].) The major vehicle for portability was a very basic subset of Fortran (ANSI-66) commonly used. The tool used to test for portability, from a Fortran language consistency point of view, was PFORT, developed at Bell Laboratories [3-10]. At first, writing even simple routines (without I/O) in a form that could compile on a broad class of Fortran compilers was difficult. Since, computers in this time frame were mostly monolithic from a computer architecture perspective, "good" algorithms generally minimized the number of floating point operations. A good algorithm on one computer, was generally a good algorithm on another. However, computer environments from a numerical perspective were different. Different word lengths, different exponent ranges (numerical ranges), and a host of seemingly minor machine characteristics differed. Even though the same Fortran source would compile on many machines, the resulting executions would many times yield different numerical results.

This common problem of execution differences was an early indication of the sensitivity of algorithms to hardware architecture, even in a purely scalar environment. The solution to this anomaly, was to interface the algorithm development to a set of machine characteristics defined by hardware environment subroutines. These subroutines did nothing more than define the differences between computers in a standard way to allow the programmer (algorithm developer) to access in Fortran callable subroutine form the very machine characteristics that could effect the algorithm robustness. The kinds of parameters defined include the following:

Symmetric range - the largest positive number x, such that x, -x, 1/x and, -1/x are floating point numbers.

Overflow threshold - the largest positive x such that x and -x are floating point numbers.

Underflow threshold -the smallest positive x such that x and -x are floating point numbers.

Relative precision - the smallest positive floating point X such that $1 - X < 1 < 1 + X$. Also called machine epsilon.

Radix - the base of the floating point number system being used. (e.g. hex, octal, or binary)

Mantissa length - the number of radix digits in the mantissa of a floating point number.

Exponential range

Assorted constants - e.g., pi, e

These and several other constants affect algorithm robustness and must be defined for single and double precision. For example, if one were advancing the time step in a Runge-Kutta method (for solving a differential equation numerically) and the current time, T was very large while the time step H was very small, it is possible that T + H = T in computer arithmetic. Knowing and employing the machine epsilon in a relative precision test could prevent this anomaly on a variety of machines.

In essence, the portability of algorithm based subroutines was achieved using an interface to the hardware characteristics of the computer. We have discussed at length the increased sensitivity of algorithms in modern hardware due, not to numerical environmental considerations, but due to fundamental architectural sensitivities that go well beyond the Fortran language. Is portability a reasonable goal, and is transportability practical in supercomputer environments? The meaning of "reasonable" and "practical" deal with efficiency. The ability to write code that functions in various computer architectures is not difficult. However, to have it perform optimally fast on several supercomputers is a much more difficult challenge. However, with years of exposure to vector processing, experienced algorithm developers, scientists, and hardware vendors have shown that, for the class of vector supercomputers, it may be *possible to develop an algorithmic approach* to this "vector optimization" challenge. In the next section a discussion of computational kernels is given and an approach to transportability of software is suggested. The question that remains open is, can an approach to transportability be developed for classes of parallel computers (e.g. shared memory machines) with reasonable efficiency in the resulting program.

************ **IMPORTANT CONCEPTS** ************

- Application programs often outlive hardware systems; thus, the topic of migrating software from one environment to another has long been a problem, even in scalar computing.

- For scientific codes, the hardware environment has traditionally been the area of incompatibility (e.g. word length, machine epsilon, single/double precision.) Vector processors have introduced a whole new level of incompatibility, the architecture which, if ignored, can cause tremendous performance degradation.

****

3.4.2 Computational Kernels as an Approach

The purpose of innovation in computer architecture is to achieve the maximum production of useful computation given a certain level of hardware (chip and circuit) sophistication. Thus, software development or migration methodology should also have efficiency as a primary goal. In this section, a discussion of a high performance approach to supercomputing is developed. At the same time, this methodology strives to maximize the current investment in computer applications and promote a certain degree of transportability among scalar and vector computers. The reader is cautioned, however, for there is no pat solution for mass software migration to new computer architectures. In any methodology of this type, a prime understanding of the application area is fundamental to success. In subsequent chapters we focus on computational fluid dynamics. Specific problem classes can be examined along with the underlying computational core of the solution. Here, a generic look at computational kernels of *common* use throughout scientific computation is offered.

Certain basic principles can be observed from a very simple, yet important computation, the matrix multiply. This operation has a great deal of structure from a data flow perspective. Two matrices reside in memory in some fashion, usually by successive columns. Since the basic operations involve rows and columns combined by many inner-products, the computation should exercise vector computers efficiently. Nevertheless, one can quite often uncover serious computational bottlenecks by performing this operation in several ways. The basic Fortran loop for multiplying a matrix A times a matrix B, to result in a product C, is given by the Fortran loop below:

```
      DO 10 I = 1, N
         DO 10 J = 1, N
            DO 10 K = 1, N                          (3-2)
               C(I,J) = C(I,J) + A(I,K) * C(K,J)
   10 CONTINUE
```

This form of the computation is the natural Fortran implementation resulting from the usual linear algebra description of matrix multiply based on inner products. Like many nested loops, this same set of operations can be performed by alternative nestings of the DO-loops. Two other orderings are given as follows:

```
      DO 10 J = 1, N
         DO 10 K = 1, N
            DO 10 I = 1, N                          (3-3)
               C(I,J) = C(I,J) + A(I,K) * C(K,J)
   10 CONTINUE
```

```
      DO 10 K = 1, N
         DO 10 I = 1, N
            DO 10 J = 1, N                          (3-4)
               C(I,J) = C(I,J) + A(I,K) * C(K,J)
   10 CONTINUE
```

In Loop 3-2, the inner-most loop is accessing a row of one matrix and a column of another. The result is accumulated as a scalar, C(I,J), for fixed outer-loop values of I and J. This is a most demanding operation for even the best vector computers. First, two vector fetches are required, and the collapsing sum is not always an efficient operation on vectorcomputers. Any operation that operates on vectors to achieve a scalar result, can be suspect, *with respect to efficiency, on many machines.*

In contrast, Loop 3-3 has the SAXPY operation as an inner-loop. As has been observed, on machines with 3-paths to memory, the SAXPY computation can be quite efficient. However, the nested loop has a further advantage. Looking at the variation of the middle DO-loop over K, one observes that *C(I,J) for I + 1, N is used repeatedly for each* value of K. In fact, this vector can be viewed as an "accumulation vector" for the K individual SAXPY operations. Thus, the final store and fetch of C(I,J) in the inner-loop is not required, and (modulo the vector register length) C(I,J) can be held in the registers. (Of course, this assumes a register interfaced architecture is being used.) This is, in fact, the supervector approach discussed in Table 3-2. Thus, Loop 3-3 can be an appropriate nest of loops for, say the CRAY-1. A further observation is that this loop can also be used well in a parallel environment by assigning each K loop to a different processor.

The third loop, 3-4, again has a SAXPY inner-loop. However, one can observe that C(I,J) is a matrix accumulation of the Kth column of A in an element-wise product with the rows of B. The result is that vectors of length N*N are manipulated by storing the repeated columns of A and repeated rows of B as large N*N vectors. While this may seem somewhat esoteric, if the start-up times for a vector operation were extremely long, this method could have an advantage over the supervector approach of loop 3-3, at the expense of extra storage.

Actually, the loops can be revisited in slightly different ways by inverting the order of access to vectors from rows to columns or columns to rows from the orders presented. The resulting algorithms would be row ordered. It is always preferable to access data by columns whenever possible in column-stored matrices. Figure 3-18 displays the near optimal performance of these loops on the CRAY-1. The differences are not due to compiler efficiency, but due to the relationship between the architectural bottlenecks and the data access flow. It is interesting to note that Loop 3-2, the poorest performer, is typically the preferred loop on a machine like an IBM mainframe, for it is equally fast in scalar form and tends to minimize page faults by accessing data in a very "expected" fashion. Loop 3-4 could be improved, but only at very non-trivial expansion of nested loops into matrices manipulated by vector instructions of length N*N.

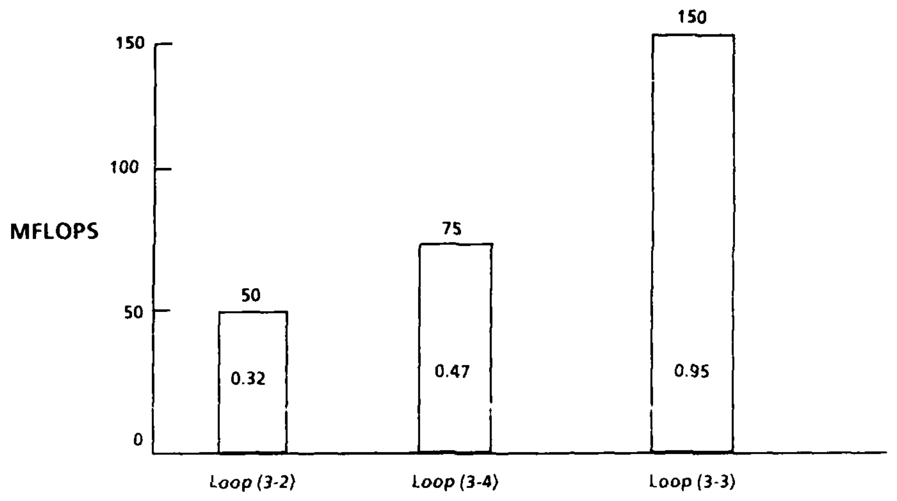


Figure 3-18
Matrix Multiply - CRAY-1

A simple observation is appropriate. If Loop 3-2 is employed in an application program, should the user alter his Fortran in migration to the vector world? The simple answer is yes - in order to achieve a possible factor of 3 improvement in performance. From a methodology perspective, however, this could be simply committing a mistake for the next round of architectural change when another loop may be appropriate. A suggested approach is to develop and migrate software at a higher level. The matrix multiply should be a module appropriately coded for the target architecture in an appropriate manner. The premise is that a computation as sensitive to architecture as a matrix multiply should be treated as a machine environment routine and handled like the square root or exponential function, which is "called" from application programs rather than coded in Fortran by the user. In this case, the appropriate routine could be implemented in Fortran or assembler. If the interface is standard, the implementation can be done however appropriate. [Note: Another approach would be to delegate the problem to the compiler writer. That is, automate loop renesting at the compiler optimization level. This may be plausible for the matrix example given here, but this is not likely to be possible for the broad range of algorithm and architecture sensitivities. For example, the likelihood that a compiler can recognize a repetitive scalar loop and transform it into a parallel/vector simultaneous implementation is very small. Certainly, this would be difficult with languages like Fortran, Pascal, or even Ada. If a new form of language was developed, where parallelism is built into the syntax, a possibility of effective optimization could exist. Even with the best of luck, such a language evolution and adoption would take many years to be adopted by high end computer vendors. For this reason, most of our comments will be confined to Fortran-like languages.]

The matrix multiply is perhaps the easiest example to discuss in such detail, since the algorithm is so accessible to scrutiny. However, many basic computations of science are equally sensitive to architecture and implementation. Another example on the CRAY-1 is displayed in Figure 3-19, a two-dimensional Fast Fourier Transform (FFT). Fortran only, Fortran with the inner computation a 1-D FFT in assembler, and a completely optimized (in assembler) version are compared. The latter algorithm maximizes supervector performance where ever possible. As displayed, the performance differences can be substantial.

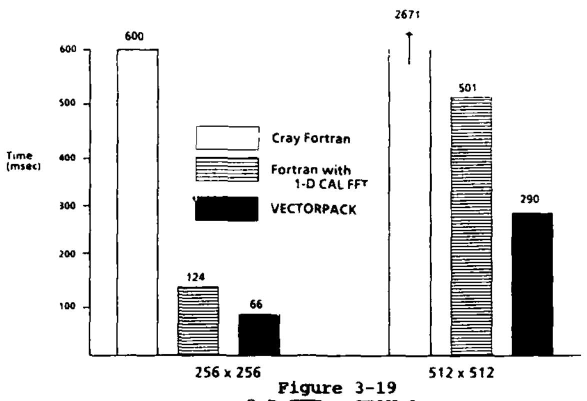


Figure 3-19
2-D FFT - CRAY-1

Again, it is emphasized that these differences are not solely compiler related, but rather due to architecture and algorithm relationships. As a further example, consider FFT algorithms on a Fujitsu VP-200 (Amdahl 1200). In Fortran, the 2-D (radix 2) FFT has been recorded at 242 MFLOPS for certain large dimensions [3-11]. The maximum value in Fortran will not likely exceed 267 MFLOPS for any size. The reason for this is simple, the Fortran standard for complex numbers is to store real then imaginary parts consecutively in memory. The FFT algorithm requires the real parts and imaginary parts to be accessed separately. Thus, at the very best, data will be manipulated with a stride of two or more. On the VP-200, this requires a loss of one path to memory. (This is an anomaly of the architecture.) However, if the FFT were defined by real arrays storing real and imaginary parts, respectively, the performance could be greatly increased. Again, what is required is a general standard interface and manipulation of data to be handled in whatever fashion is best for the target architecture.

The above discussion supports the following premise:

Premise 1. Architectural sensitivity in computation can transcend the domain of the Fortran compiler.

Probably the most striking way to illustrate the complex relationships among architecture, algorithms and compilers, is to hold as many variables fixed as possible. To that end, we will examine seven computational kernels (i.e., simple one or two-level Fortran loops) on the CRAY-1 and the CRAY X-MP. The kernels are given below:

1. SAXPY
2. CAXPY - same as the SAXPY only in complex form
3. Sparse SAXPY - Figure 3-8
4. Inner product (dot product)
5. ISAMAX - find the maximum component of a vector
6. Matrix multiply
7. Complex FFT (entire algorithm)

In order to display performance, the two-dimensional domain (introduced in Chapter 2) will be used. Figure 3-20 displays the near optimally coded kernels for the CRAY-1. It can be observed that Loop 1 is inferior to loop 6 for a dense vector operation (for reasons described earlier). Loop 3 is observed to be nearly scalar in both its peak performance and its relatively small $N_{1/2}$. Nevertheless, Loop 3, which was discussed at great length earlier, is coded roughly 3 times more efficient than the compiler version. It is also worthy to comment that Loop 5 suffers from being a vector to scalar type loop riddled with compare operations. Loop 5 is often employed in pivoting schemes in matrix decomposition algorithms. Loop 6, obviously coded optimally, displays the high performance achievable with the "right" matrix multiply. This is three times more efficient than the usual Fortran.

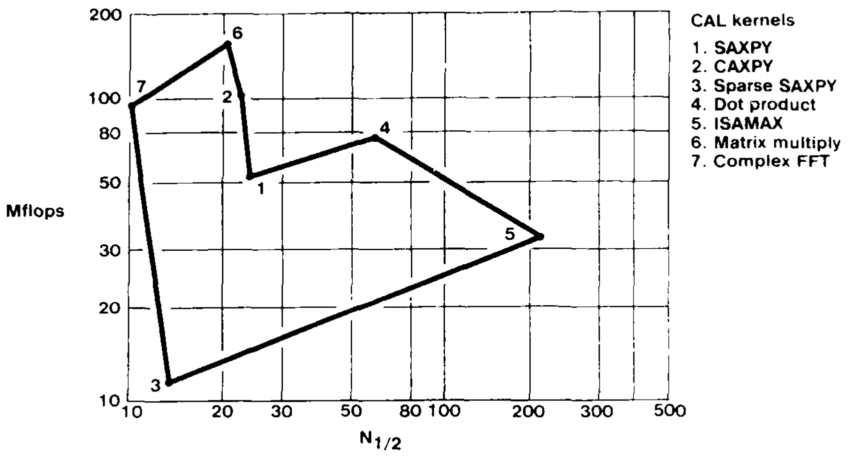


Figure 3-20
Seven Kernels - CRAY-1

Figure 3-21 displays the same loops on the X-MP line. There are several X-MP types. The older X-MPs did not have gather/scatter operations, consequently the sparse loop (Loop 3), performs almost as poorly as on the CRAY-1. The newer X-MPs (9.5 nanosecond clock) has loop 3 performing at a much higher rate with a larger $N_{1/2}$, 90 MFLOPS and 40, respectively.

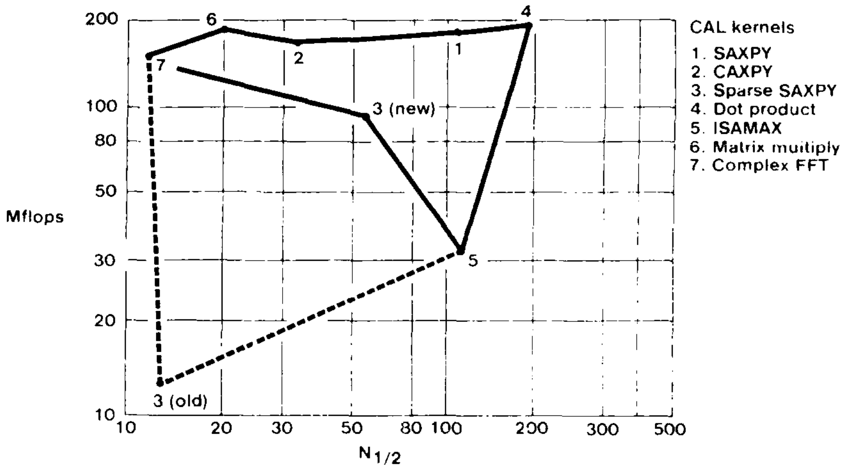


Figure 3-21
Seven Kernels - CRAY X-MP

Figure 3-22 displays the CRAY-1 and X-MP for comparison. Note the startling difference between the new X-MP and the CRAY-1. The only difference in cycle speeds is the 25% faster X-MP clock. Yet, the performance differences are significantly greater. Note that loops 1, 2 and 4 have become very high performers on the X-MP, while they were poor on the CRAY-1. This leads to the following premise:

Premise 2. A "good" computational kernel on one computer may be a "bad" kernel on another.

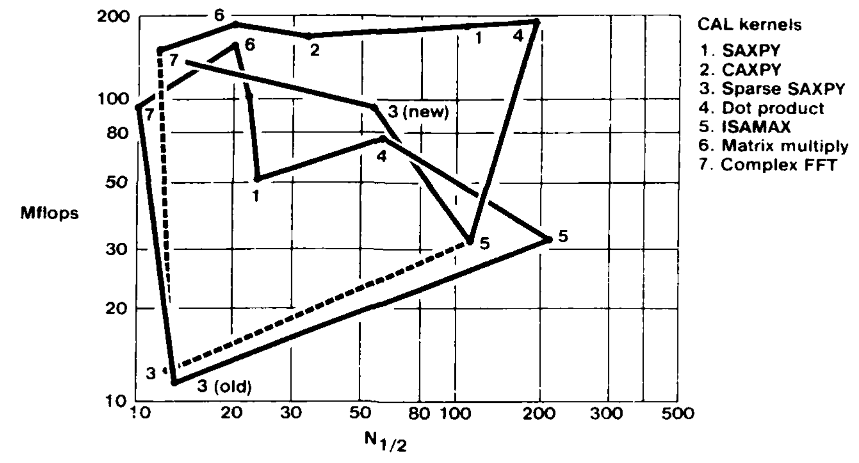


Figure 3-22
Seven Kernels - CRAY 1, X-MP Overlay

Recall, that in all three kernel figures, the optimal algorithm was selected and coded in Cray Assembler Language (CAL). In Figure 3-23, the X-MP CAL code performance polygon is overlayed with the Cray Fortran (CFT 1.13) implementation of these same algorithms. The 1.13 compiler has been substantially improved over the years by 1.14 and 1.15 versions. However, CFT 1.13 was the best available for a number of years. In fact, some applications have faster execution speeds on the 1.13 compiler than the newer compilers. These applications probably are not benefitting from the superior optimization capabilities of the newer compilers. From Figure 2-23, one can easily note the poorer performance of the compiler over assembler. In fact, loops 3 and 5 are not even on the chart. This is not surprising at all. It takes over 5 years for a compiler to mature. In 5 years a new architecture is obsoleted by a new one. Therefore, one can conclude that if the compiler is efficient, the machine is obsolete! These facts lead to the following conclusion:

Premise 3. Compilers can seriously degrade potential performance on newly introduced computers.

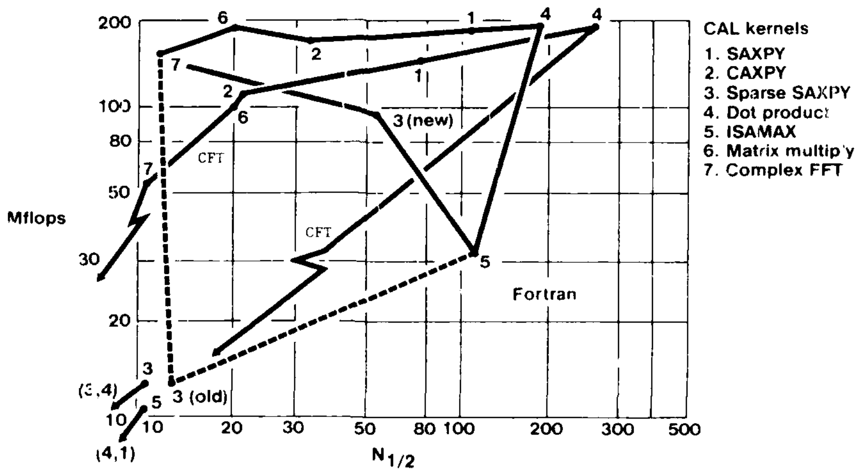


Figure 3-23
Seven Kernels - CFT 1.13

What, then, are the implications of the premises proposed?

Premise 1. Architectural sensitivity in computation can transcend the domain of the compiler.

Premise 2. A "good" computational kernel on one computer may be a "bad" kernel on another.

Premise 3. Compilers can seriously degrade potential performance on newly introduced computers.

One principle seems clear. Total dependence on Fortran (and other sequential compilers) compilers as a means of program optimization is not wise. Since software production programs survive several compilers in various stages of maturity, the reliance on compilers for performance is bound to result in varying performance results. Since proper algorithm selection is dependent on sound computational kernels, and since the proper kernels may be different for different machines, the only hope is to interface to the hardware much higher in the complexity of the modeling process than available at the compiler level.

In designing applications programs on modern computing machines, there are often stages of modeling and implementation that result in a "running code." This modeling hierarchy proceeds from the physical model to the computer object code in all or some of the following stages:

STAGE 1:	THE PHYSICAL PROBLEM
STAGE 2:	MATHEMATICAL MODEL
STAGE 3:	COMPUTER MODEL
STAGE 4:	SPECIFIC ALGORITHMIC APPROACH
STAGE 5:	THE PROBLEM INPUT MODEL
STAGE 6:	THE FORTRAN IMPLEMENTATION
STAGE 7:	SYSTEM OR OTHER COMPILED LIBRARIES
STAGE 8:	ASSEMBLER AND OBJECT CODE

A lengthy description of each stage will be avoided. The reader can probably fill in details from past experience. Basically, the physical world is being modeled by a set of mathematical laws which are cast in numerical (discretized) algorithms. Many times existing modules (such as linear equation solvers) are used as tools in building the computer model. The problem definition, itself a data model, will be abstracted by input procedures. For example, in a computational fluid dynamics code, the wing or airplane must be modeled by surface approximations, which in turn interact with the computational model. The final implementation is done through a higher level language, most probably Fortran. The Fortran compiler uses tools such as libraries for elementary functions and the like. The final output from the compiler is a very machine specific set of instructions in assembler or object code.

For many years the impact of changing the computing hardware was confined to stages 6, 7, or 8. In other words, the accommodation of a new computer into the modeling hierarchy generally had little impact "above" the Fortran level. A new computer meant that the application programmer would have to convert to a new brand of Fortran. Once converted, most of the impact of the new hardware was accommodated by the operating system and/or compiler. Now, in dealing with supercomputers, the issues of efficiency go well beyond the confines of the compiler. Moving to a new computer can have, and has had, impact in all stages of design. Even if the the physical model isn't changed, the algorithms used, the mathematical models, and range of the physical model parameters are often impacted. Accommodating new hardware was once the domain of the operating system and compiler and is now a more fundamental issue. To whatever extent this is a problem today, it will be accentuated by the increasing use of parallelism in hardware design in the future.

These facts suggest a new approach to computer code design that attempts to transcend the confines of Fortran when developing or converting existing programs to vector processing. Figure 3-24 illustrates the concept of moving the boundary of the interface between models and hardware higher in the complexity of the modeling hierarchy. This is done by relying on a set of standard computational building blocks, or kernels, that have two properties:

1. They are extremely important in application programs in that they often consume most of the CPU time in the algorithms or processes that use them.

2. Their performance is extremely sensitive to architectural features.

The basic approach is to build the application model from higher level algorithms whenever possible. Standard libraries of tried and true computations exist at most locations, industrial and academic. (For a rather thorough description of quality library software see [3-7].) The next step is to provide, to both the library developers and the applications programmer, a set of optimized computational kernels. The important feature of these kernels is not only speed but standardization. They should exist on all computers being used for scientific computation whether scalar or vector.

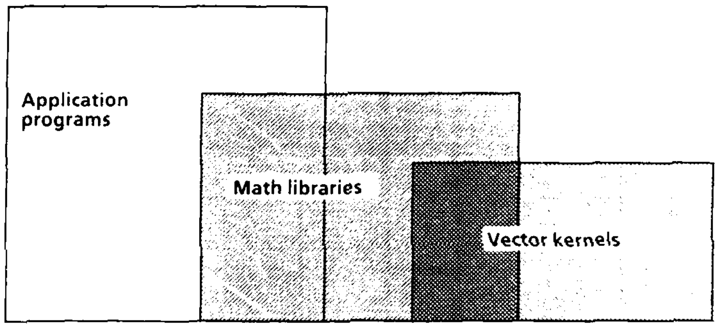


Figure 3-24
Moving the Hardware/Software Boundary

In addition the functionality of such kernels in new hardware environments should be assured. This could be done by having two versions, one optimized for a particular machine, and another in a portable language (such as Fortran or C). These interfaces should be standardized, and they should be coded in an optimal form using the best computational strategies for the various target computers. This is not something scientists should be burdened with producing. In fact, this is no different than the sine or cosine routines provided by the compiler. There are very few scientist today who can, or wish to, code appropriate and accurate algorithms for elementary functions, yet all scientists use them liberally. The community of scientist exploring computation in CFD, for example, should begin to identify and define important commonly used computational algorithms and underlying kernels that are critical to common computation. This will be discussed further in Chapters 4 and 5.

In practice the number of standard frequently used computations in science and engineering is not that large. In later chapters kernels important to computational fluid dynamics are discussed. By interfacing complex processes such as algorithms or entire solution modules to basic architecturally sensitive computational kernels, code developers can concentrate more on modeling, and computer/algorithm scientists can concentrate on the proper implementation of kernels. The methodology is being used in a number of industrial locations. The shortfall is that not all major application programs are written in algorithmic modular form. Modularization is becoming more of a necessity, however, due to the rapidly changing computer environment.

There are many examples of the efficacy of this approach. One example, has been discussed in detail. Recall Table 3-5 in Section 3.2.2 which shows a set of matrix problems whose performance improved by a factor of 3 over the CRAY-1 performance due to the improvement of the sparse SAXPY computation on the two machines. The algorithm software had been "interfaced" to the assembler coded sparse SAXPY kernel. This kernel had been developed and optimized separately. On conversion of the algorithm to the X-MP, nothing was done except to replace the kernel library called by an optimized X-MP version. Had this been done in Fortran, the performance on the X-MP being used for these tests, would have yielded only a 25% performance improvement in the kernel and negligible improvement in the algorithm. Why? At the time, the test was run, the latest CFT compiler was 1.13 which did not utilize the important "new" gather/scatter instructions. Thus, the code optimization "transcended" the compiler optimization level. Even new compilers have trouble "recognizing" the sparse loop, and often must be directed to vectorize them due to possibility of recursion which only the user can rule out.

Another example of the possible improvement to entire programs is given in Figure 3-25. The application is a large Kalman filter modeling program used in guidance and control systems. It was divided into three computational sections and a large input/output section. In fact, only 10% of the lines of Fortran accounted for the over 95% of the performance. This is indicated in the figure by bar labels: matrix operations, transition matrix, and noise matrix. The other 5% of the computer time was spent in the section labeled other. The percentages given at the top of the bars in the graph, indicate the percentage of CPU time was spent in the module by the original program fully optimized by the Fortran compiler (CFT 1.11 on the CRAY-1.)

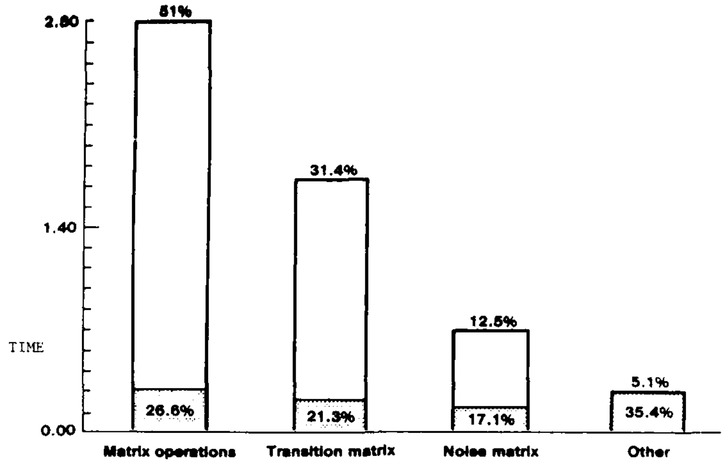


Figure 3-25
An Application Program

The shaded bars indicate the percentages after the code was enhanced by replacing Fortran coding with optimized kernels from Vectorpack [3-14]. This replacement took a few days labor in order to isolate the computations in a rather unstructured Fortran program, typical of production programs. The result dramatically reduced the CPU time in each major computational module. The entire program ran 8 times faster than the original Fortran optimized program. This program was optimized via computational kernels. Several years later it migrated without change to a CRAY X-MP where these same standardized kernels were provided, but optimized to the X-MP. Consequently, this optimal, better than Fortran performance, was maintained without user intervention.

************ **IMPORTANT CONCEPTS** ************

- In the examination of important computational kernels, three empirical observations were supported:
  - Architectural sensitivity in computation can transcend the domain of the compiler.
  - A "good" computational kernel on one computer may be a "bad" kernel on another.
  - Compilers can seriously degrade potential performance on newly introduced computers.
- In scalar computers, hardware evolution was largely the domain of chip technology, causing little impact in applications beyond the compiler and operating system. In vector computers, the impact of subtle architectural features can impact computational kernels, algorithms and even entire applications. This, coupled with the rapid growth in computational power, can have ramifications at the modeling level as well.
- It has been observed that compilers seem to take 3-5 years to mature. It also can be observed that supercomputers are obsolete in 3-5 years. One can conclude that compilers of the Fortran variety cannot be relied upon, solely, as a tool for optimization.

****

3.4.3 The Role of Languages, Compilers, Preprocessors

The general topic of computer languages is well beyond the scope of this treatise. The attributes of a "good" computer language are debated continually. In engineering/scientific computing the dominant language force has been, and for the foreseeable future, will be Fortran. At this point, the dominance of Fortran has little to do with its merits, and more to do with its position as the support language for programs that fuel scientific technology world-wide. The person years of investment for development and maintenance of this software is difficult to estimate, but is enormous. For a major oil company or aerospace company to convert its production application programs to another language would be cost prohibitive. It is not likely any language or computer/compiler combination could prove to be so valuable as to cause such a conversion. Even if a conversion is undertaken, large production programs are only as useful as they are reliable. Newly converted programs lose that validation from the tests of repeated daily usage. This will always be the biggest deterrent to code optimization. New languages can become useful as new programs are developed in them. Yet no language seems to have achieved a substantial foothold in new development projects of import. This situation may change. In fact, if we are ever to aspire to harness massive or large degree of parallelism, it is likely that new languages could hold the solution. However, such a process could take decades, and the current issue is how to effectively compute on today's machines. For these reasons, this section will predominantly deal with Fortran, its extensions, and preprocessors. In fact, when the term compiler is used, it will generally refer to the Fortran compiler.

Historically, the higher level language was a method of programming the computer through a more user friendly "language" than the machine language or assembler (which are simply the instruction sets of the hardware). The efficiency of the language was a function of the translation of the higher level syntax into the most efficient sequence of assembler instructions where efficiency was essentially measured by maximizing the instructions per second. Algorithm efficiency on scalar computers was simply a matter of minimizing the number of adds and/or multiplies in the process. The algorithm developer and the compiler optimization process were rather separate entities (with the exception of the obvious overlap of elementary functions, e.g., SIN X.) With the advent of pipelined arithmetic units, and more recently, vector instructions the two fields have become very intimately related. On the one hand, armed with some knowledge of how the compiler may, or may not, assign vector instructions to a syntax, the algorithm developer can "write" very efficient Fortran on a vector computer. On the other hand, with some knowledge of the "intended" use of certain syntax, the compiler writer can achieve a tremendous improvement in instruction sequencing or vector instruction selection. The natural outcome of these facts has been a growth of appendages to Fortran such as the following:

Vectorization directives
Vector syntax
Interactive compilers and user directed optimization
Preprocessors

It is important to understand how these artifices work, for they can indeed improve performance. One must realize, however, that at best these are attempts to accommodate the fact that Fortran cannot recover from algorithmic and architecture conflicts. The compiler can aspire to achieve the highest efficiency for a given computational kernel. If this kernel is a poor computational tool for architectural reasons, the compiler or extensions cannot compensate. Ultimately, optimization rests in the hands of the code developer. It is not unlikely that in certain situations the compiler can achieve several factors improvement in performance through optimization techniques. It is, however, possible that a factor of 4 or 8 can be achieved by simple kernel optimization and even more with modeling changes.

In Section 3.3 the term "vectorization" was defined in terms of percentage of CPU time spent in vector unit execution. Compiler optimizers generally must deal at the instruction level and optimize performance based on the premise that vector instructions are to be substituted for scalar instructions whenever possible (with perhaps an identified set of exceptions). More modern compilers are attempting to involve the user through interaction providing statistics and asking questions to gain more information than can be discerned from the syntax. Finally, some compiler efforts are beginning to make compilers that implement decisions at run-time in an effort to test on data parameters that are not usually known at pre-execution. It is perhaps worthwhile to give a few examples of straight forward optimization capabilities, more elaborate optimization, and predictions of future compiler optimization capabilities.

Supercomputers rely on vector operations for pipeline or parallel computation efficiency. This is not an accident, but a result of the observation that scientific programs often use vector constructs in Fortran. The most common is the simple DO loop.

```
      DO 10  I = 1, N
          X(I) = Y(I) + Z(I)                    (3-5)
   10 CONTINUE
```

This loop is equivalent to a simple hardware vector instruction on most vector computers. It is easy to "recognize" and easy to systematically translate into a vector instruction. In fact, most compiler writers have little trouble in translating this loop to optimal code for the target architecture. Unfortunately, Fortran application programs are often contain optimization prohibitors such as IF-tests, subroutine calls, GOTOs and recursion. Many compilers simply evolve with time and hardware vendor experience to recognize patterns and exceptions/alternatives to the standard compilation process to improve the optimization results.

The basic tool which compilers use is dependency analysis. If vector inputs are independent of the vector result, a vector operation can be employed. The historical approach has been pattern matching, but more recent compilers are using algorithmic tools to discover and analyze dependencies. An obvious dependency is recursion:

```
      DO 10 I = 1, N
          X(I-1) = Y(I) + X(I)                    (3-6)
 10   CONTINUE
```

This loop is not vectorizable because of dependency of the output on the input. (Note: the NEC machine can vectorize this loop. Therefore, for that machine simple dependency analysis does not rule out the possibility of further optimization. Further analysis can be done.) It is not difficult to imagine an example where the compiler would really benefit from some user knowledge that cannot be discovered analytically. For example, consider the loop:

```
      DO 10 I = 1, N
          X(I+J) = Y(I) + X(I)                    (3-7)
 10   CONTINUE
```

The vectorizability of this loop is dependent on the variable J. For example, if J=0 this loop is vectorizable in the traditional sense. However, if J is, say, larger than N, this loop can be vectorized. For this type of situation, many compilers today allow for user directives to "force" vectorization. Many compilers also inform the user which loops can or cannot be vectorized, and some compilers even allow for tuning during an interactive compilation processes. Probably the most publicized compiler optimization in supercomputing has been the so-called IF-test optimization introduced by the Japanese manufacturers, first by Fujitsu. One of the more frustrating constructs to compiler optimization process is the IF-test. This instruction pervades common Fortran programs. The following is an example of such a loop:

```
      DO 30 I = 1, N
          IF ( X(I)  .GT.  EPS ) THEN
           Z(I) = X(I) + Y(I)*S                 (3-8)
          ELSE
            Z(I) = 0.
          ENDIF
 30   CONTINUE
```

IF-test optimizing compilers face the same choices discussed in Section 3.2.2, where a similar situation was studied in detail. Depending on the success ratio of the IF-test several courses of action can be taken. Most supercomputers, today, have the option of mask vectors (vectors associated with bit control vectors to indicate yes/no or true/false corresponding to component position), list oriented gather/scatter operations, and simple scalar approach. None of the vector options, however, are simple vector solutions. For example, if the "success ratio" were 50%, then the loop would split into two vector loops with half the vector length. The best optimization might be to use a mask vector to split the loops and full vector computation in each loop. Thus, for the example above, there would be a vector loop (length N) to split, a vector loop (length N/2) to compute the SAXPY, and a vector loop (length N/2) to set X to zero. This is hardly a desirable optimization in the sense that a single vector loop of length N is all the complexity that may be required if the larger process were analyzed. The situation is much more complex than this. How does one know the success ratio of the IF-test? IF success ratio were almost always 100% one could perform a different type optimization. To aid in this processes some compiler environments allow the user to gather statistics, input suggested success ratios, and otherwise interact with the compiler. This, unfortunately, is problem dependent, and very little definitive optimization could really be accomplished except on a fixed problem. (After all most "benchmarks" are a fixed problem!) Viewed in this manner, IF-test optimizers are very good vendor benchmark tools, and not necessarily good approaches to program optimization.

Table 3-5 lists manufacturers and their compiler options in the area of compiler/user features. The "User Directives" column indicates that the compilers allow the user to force vectorization in areas where the compiler may not be able to determine possible independence of vector components as in Loop 3-7. "Interactive Tuning" indicates that the compiler allows user interaction during the compilation process, and provides information to the user about possible optimizations which require more information about. "IF-test Optimization" is a compiler vectorizing feature that provides some form of vectorization of DO-loops with embedded IF-tests. This is more effective if the compiler also allows user interaction in the optimization of these IFs (subject to the caveats previously discussed.)

Compiler Optimization

MANUFACTURER	USER DIRECTIVES	INTERACTIVE TUNING	IF-TEST OPTIMIZATION
CRAY	yes	no	yes
CYBER 205	yes	no	no
FUJITSU	yes	yes	yes
HITACHI	yes	yes	yes
NEC	yes	yes	yes
IBM	no	yes	yes

TABLE 3-6

As compiler vectorization becomes more sophisticated, compiler optimizers are looking beyond the simple inner-DO-loop. For example, on several machines we have discussed, there is a degradation due to non-unit stride. [On the IBM 3090/VF the degradation is due to a potential increase in cache hits. On the CYBER 205, it is due to the hardware instruction itself, which is defined for contiguous vectors only. For the Fujitsu VP-200, a stride operation loses a path-to-memory. The CRAY-1 has only one path-to-memory.] Consequently, the following loop has a number of possible optimization challenges on various machines.

```
          DO 10 K = 1, M
          DO 10 I = 1, N
             X(K,I) = X(K,I) + Y(K,I)                (3-9)
      10  CONTINUE
```

Some compilers (e.g. Fujitsu VP 200) are "smart" enough to recognize that the inner loop is ranging over the rows rather than columns. Under legitimate circumstances the loop execution is interchanged. Most programmers would not code this loop today because of increased awareness of this degradation. Another problem which compilers can address can be explained using loop 3-9. Quite often the length of vectors in the inner-loop can be very different than the vector lengths in the outer loop (e.g. M >> N). In this case the longer loop should be executed in the inner loop to achieve greater performance (i.e. operate closer to the asymptotic maximum.) This however, is a run-time decision unless M and N are known constants at compile time. In addition, for a loop as simple and pristine as loop 3-9, some compilers can recognize that the following loop is equivalent and compile one vector instruction of length M*N rather than M instructions of length N.

```
          DO 10 I = 1, N*M
             X(I,1) = X(I,1) + Y (I,1)           (3-10)
      10  CONTINUE
```

Other compiler techniques can be important, and not all compilers employ enough sophistication to use them. Figure 2-20 in Chapter 2 showed a technique for path-optimization by splitting a loop into processing half vectors rather than full vectors. If the vectors are long enough the extra number of vector startups, due to the doubling of vector instruction issues, can be negligible with the gain. This "trick" is not often employed by compilers. Nevertheless, the resulting 50% asymptotic improvement is not insignificant. Quite often compiler optimizations are merely the correction of poor coding practices. In such cases the compiler can improve performance. (Such improvements may give one a sense of false security.)

A number of loop forms are now handled by compilers. The obvious ones such as SAXPY and inner-products are optimized by most compilers to use vector instructions. The sophistication in some compilers matches the sophistication of the instruction set. For example, on the Fujitsu VP-200 the following loop is translated to efficient vector instructions available in the hardware instruction set.

```
      DO 10 I = 1, N
         VMAX = MAX(VMAX,PM1(I))             (3-11)
         VMIN = MIN(VMIN,PM2(I))
  10  CONTINUE
```

Each manufacturer will improve compiler optimization over time. Early compiler efforts were not very ambitious. As a consequence a number of commercially available preprocessors were developed. They essentially did the compiler's job better. Unfortunately, use of preprocessors sufficiently complicates the software development process, and their widespread use was generally avoided. Today successful hardware vendors devote greater effort to compiler technology and eventually out-perform the preprocessors. In fact vendor compilers eventually utilize every flexibility built into the hardware. For example, the CRAY-2 provides for local memory of 16K words. This is more of a generous register set which is easily accessed by the vector units in contiguous storage mode only. Initial CRAY-2 compilers did not use this feature. In the future it will be used more aggressively to ameliorate the effects of the single path-to-memory. Similarly, a feature like variable vector register length on the Fujitsu series, can be a tool for optimization. The initial compiler efforts set the register length to a fixed length, say 512. If one were processing a loop of length 600, this would be done in two slices, one of 512, and one of 88. Perhaps this could be optimized by performing three slices of length 200, or by one slice of 600. In each case the registers must be reconfigured which halts all processing for one or two cycles. Thus, the benefit of reconfiguration has an overhead.

What can we expect from compilers in the near future? The biggest near term improvement will probably come from run-time optimization. This has been alluded to several times. Quite often, it is not until execution that certain facts about loop characteristics are known. At the small expense of longer compilations, and space for compiled options executed according to run time decision, a tremendous improvement in performance could be attained. The biggest language challenge of the future will be parallelism which promises to be much more difficult for traditional Fortran to accommodate in an automatic manner.

************ **IMPORTANT CONCEPTS** ************

o The sophistication of optimizing compilers is increasing

- Simple loop optimization is common place.

- *Nested-loop optimization is being performed by some compilers.*

- Interaction with the user is being provided enabling the compiler to gain more information about the program, and enabling the user to learn where the compiler is having trouble optimizing.

- A likely new area of improvement will be run-time checks of critical data required to optimize. This data, such as loop parameters, is not available at compile time for it is problem depended.

o Compilers cannot make algorithmic decisions!!!

****

3.4.4 Throughput and I/O

Up to this point we have concentrated on the internal function of the computer. Experience dictates that the value of a supercomputer is really a function of "turn around." One often hears of users who are willing to have a program run days or weeks if necessary to get a new result. These types of users are rare in the total economics of computing (super or otherwise). Most technological programs are required to aid in a bigger process, such as research or design, where the human (the scientist or engineer) is an integral part of the process. For the most part there is a fundamental period of response time for which results are required. For example, if a scientist cannot get results within a 4 to 24 hour period, he or she will tend to simplify the computation. To study throughput, then, is to study the total turn around time of a job. There are a number of factors outside a single user's control such as the job stream and queuing algorithms employed. These topics will be avoided here, but are of specific interest with respect to parallel CPUs. This will be discussed briefly in the next section. For a single job, one can divide the processing into the following steps:

Input
Execution
Output

Quite often an inordinate amount of overall time is spent in input and output. This can be dealt with in many ways. As computing sophistication increases, more of the input model will be automated and less of the output in raw data will be required. The real issues are within the execution phase, an area that has been traditionally disassociated with I/O. We are speaking of data movement associated with calculation. In subsequent chapters, it will be observed that an important performance issue in algorithms for computational dynamics is memory and data management. The data required to define the *mathematical discretized problem (with boundary data) is often dwarfed by the* intermediate data generated during the computation. For state of the art problems the combined memory data requirements often exceed the largest memories. Since conventional disk technology is too slow for modern supercomputers, a real performance bottleneck can arise. To illustrate the range of performance improvement that can be achieved with faster secondary storage, an example from structural analysis is given.

In Chapter 2, Section 2.2.3, the notion of secondary storage was discussed relative to disk storage speeds. Consider a CRAY X-MP/24 (i.e. a 2-CPU system with 4 MW of real memory) with an additional 128 MW-SSD. A traditional approach to computing large structural analysis problems, is to use a direct matrix factorization algorithm that takes advantage of limited real main memory. (See [3-12].) This has been a traditional approach in this field enabling the solution of problems that require more data than can fit in the real memory available. The intermediate data generated is written to disk or secondary storage, if available. The statistics, given Table 3-7 reveal the effectiveness of having a fast secondary storage device even if it is volatile.

Data Movement and Throughput: An Example
Structural Analysis - CRAY X-MP/24

	X-MP NO SSD	X-MP WITH 128 MW SSD
I/O DISK	1,112 MW	295 MW
CP TIME	1.6 hrs	0.7 hrs
TOTAL WALL CLOCK	19.0 hrs	1.1 hrs.

TABLE 3-7

The most important figure in the table, is the wall clock time. The dramatic reduction in this figure afforded by the use of SSD can change a job from being less than a practical research tool to a few hour investigation. The productivity of the user is greatly enhanced. The same is true for adding more real memory. However, when the total system memory is increased, as dramatically as it was in the release of the CRAY-2, whole new realms of modeling become practicable. The CRAY-2 will provide valuable insight for all manufacturers.

************** IMPORTANT CONCEPTS **************

o Data management is one of the most critical aspects of high speed computation. When the data exceeds available memory, I/O management becomes a critical performance factor.

o Conventional disk technology is not keeping pace with other aspects of computer technology. Thus, secondary storage has become a potential critical bottleneck in modern computing.

o A computer design issue is currently unresolved. Is the expense of large monolithic memory justified, or is a tiered memory structure adequate for most, if not all applications?

****

3.4.5 Parallelism: Top Down or Bottom Up?

Will parallelism, in the form of multiple CPUs, become a mainstay architecture in supercomputing? The answer seems to be a certain yes. The question seems to be more one of degree and timing. The critical issue is software and the ability of researchers, scientists and engineers to produce applications programs that can fully utilize this type of computing architecture. In this section we will briefly discuss software issues related to the use of parallel-CPUs in supercomputers.

The use of pipelined arithmetic units has created the possibility of improving computation by a factor of 10 or 20 over scalar performance with comparable chip technology. As software has been able to accommodate the changes required by "vector" computing, the improvement in performance has crept closer to this potential. In a similar fashion, parallelism offers a potential for performance improvement. A 16-CPU vector computer offers the combined potential of 16*20 (320) times the performance improvement of a scalar computer with comparable chip technology. To illustrate consider Figure 3-26 below. The figure illustrates ideal performance improvement from parallel CPUs with no degradation due to memory conflicts, synchronization overhead, or contention for system resources. In addition, it is assumed that ideal vector speedup is attained on 100% of the code using vector lengths well beyond three times $N_{1/2}$. Ideal though it may be, the result illustrated also depends on a critical factor, parallel decomposition of software and underlying algorithms. Once again, the promise of this potential resides in the domain of software (operating systems, compilers, algorithms and application programs).

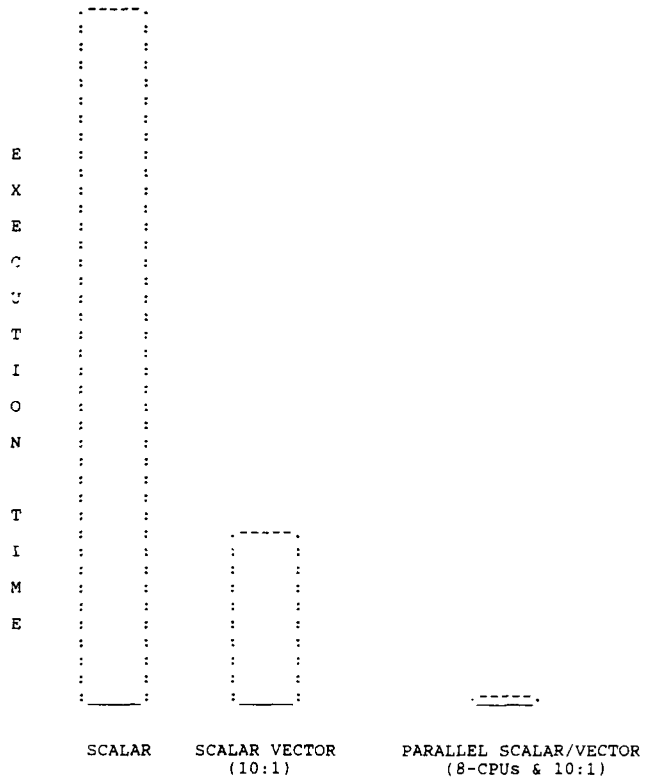


Figure 3-26
Parallel and Vector Interaction

It is useful to examine the ways in which a relatively small number of CPUs can be used to improve performance. There are three levels in which parallelism can be exploited to improve system performance.

1. The job stream level: Given the fact that supercomputers and general purpose computers often are used in a multi-user environment, system throughput can be improved by allowing different jobs to run concurrently and independently in the available CPUs. With only two (or perhaps four) CPUs, overall throughput can be improved relatively easily (without user intervention) by the operating system. Of course, this assumes that the individual jobs are not competing for shared resources such as memory or peripheral devices. It is not likely, however, that 16 or 32 CPUs could be utilized productively in this fashion.

2. The job step level: A common throughput performance improvement technique in scalar computers is to divide the job into subtasks (typically I/O, CPU, compile etc.). This can be equally useful in a parallel environment. This technique can be distinguished from user influenced actions (below) in that it can be automated at the operating system level with perhaps help from the compiler.

3. Program level multitasking: At this level performance improvement is obtained by decomposing a single program into subtasks that perhaps require cooperation (synchronization and data exchange) among the subtasks. This type of job decomposition seems to be an inexorable requirement in the effective use of multiple CPU systems from 8 to 30 CPUs.

It is this third level that is the focus of software/algorithm technology today. Two alternate, yet perhaps, complimentary directions have emerged in commercially available systems. One approach is based on asynchronous (i.e., MIMD) utilization of parallel CPUs. With this approach even inherently scalar (sequential) processes can be parallelized. This can be characterized as top-down parallelization. An example of this approach is provided by multitasking extensions to Fortran provided by Cray Research on their multiple CPU machines. Although the constructs are different, ETA Systems offers similar multitasking tools. The other approach is a more bottom-up approach, which can capitalize on highly vectorized code to orchestrate automatic parallelization at the inner DO-loop level. An example of this is given by CRI's compiler related product called "microtasking." This is similar in philosophy to what the Alliant Computer Company has done with their compiler. Both of these approaches allow for automatic parallelization at the DO-loop level. The main criticism of this bottom-up approach is that it does not really require parallel CPUs from an architectural point of view. Similar improvement could be obtained by introducing multiple arithmetic units (pipelined or scalar) into a single CPU.

The two techniques, discussed above, are not mutually exclusive and could be combined to achieve greater performance. As a hypothetical example, imagine a 16-CPU machine, with each CPU, itself, a vector architecture (e.g., a CRAY-3). One could employ a strategy of decomposition of an algorithm into four main synchronized subtasks. Quite often it is easy to recognized three or four relatively independent (and equally complex) tasks high in the program structure. Assume each subtask is generally utilizing long vector operations. One could cluster 4-CPUs to each major task using a top-down approach. Within the four clusters of four CPUs, one could employ a bottom-up approach breaking the long vector processes into four identical vector process spread across the four CPUs. The vector lengths would be one quarter the original length. (With some luck there might be a local nest of DO-loops allowing a group of vector operations to be spread among the four CPUs with no vector length degradation.) This hypothetical situation is probably quite likely to be utilized on the CRAY line. In fact, CRAY users are already experimenting with macrotasking (top-down multitasking) and microtasking (bottom-up multitasking) interaction.

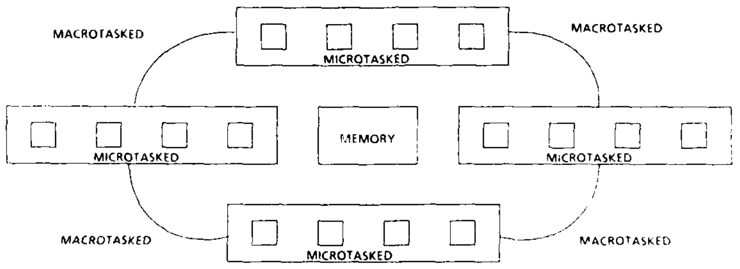


Figure 3-27
Hypothetical Example: Macro- and Micro-tasking

As simple as the above scenario is, several complex decisions must be considered. The first is Amdahl's law for parallel computers. (Refer to Amdahl's law in the glossary, Appendix A.) The fundamental penalty for parallelism of the variety being discussed, is overhead for synchronization and "wait time" due to resource contention. When these overheads are large, it becomes hard to capitalize on performance improvement unless the "granularity" is also large. The term granularity refers to the "size" (measured in time) of the subtasks. We have previously alluded to the overhead of vector operations due to the startup time. Vector length is analogous to granularity. The longer the vector, in a sense, the greater the vector task's granularity; consequently, the more productive the vector operation, since the overhead of startup time is amortized over a larger time slice. In a parallel environment, the overhead for task starts is constant. Therefore, if a computational task is being performed by 50 tasks, the overhead is 50 times a single task start. Thus, one would strive to have the computational task being performed large enough to absorb this overhead. With these two overhead concerns consider the following Fortran loop:

```
          DO 10 I = 1, N
          DO 10 J = 1, M
              A(I,J) =  A(I,J) + B(I,J)                  (3-12)
      10  CONTINUE
```

In a parallel vector environment, a number of options exist. First, the loop could be executed as follows:

```
          DO 10 I = 1, N*M
             A (I, 1) = A (I, 1) + B (I, 1)            (3-13)
       10 CONTINUE
```

In fact, a good vector compiler will make this optimization. The parallelization of loop (3-12) could be accomplished by spreading the outer-loop to the available processors each of which would have the inner-loop available for vectorization. Several questions come to mind:

> What if the overhead for so many processes is too great?
> What if both microtasking and macrotasking are available?
> Which should be used? Or should both be used?
> What if M is very small and N is very large?

Some of the answers are fairly obvious once overhead characteristics are known for both vector, macro-, and micro- processes. These considerations will be fundamental to the discussions and examples in chapters 4-6.

Impact on Applications

The modest parallelism found in current generation CRAY computers has offered early experience with parallelization of large application programs. More is being learned from mid-range parallel processors such as Alliant, ELXSI, Sequent and others. However, CRAY machines are running the most ambitious large-scale application programs. Chen [3-13] offers the following statistics.

Application Performance

APPLICATION	PERCENT PARALLELIZABLE	(4-CPU CRAY X-MP) THEORETICAL SPEEDUP	ACTUAL SPEEDUP
PARTICLE-IN-CELL	97%	3.67	3.48
WEATHER FORECAST	98%	3.77	3.55
SEISMIC MIGRATION	98%	3.85	3.45
MONTE CARLO	99%	3.85	3.75
LATTICE GAUGE	100%	4.00	3.77
SEISMIC	98%	3.80	3.50
AERODYNAMICS	99%	3.86	3.60

TABLE 3-8

These figures hold promise for the efficacy of parallelism. Some of the above applications are notoriously easy to parallelize. Yet, the parallelization is done by hand. Only four CPUs are used, and yet the degradation due to synchronization overhead is already apparent in several cases (notably in the oil and aerospace applications). What is the likelihood of automatic parallelization? What can be expected from 8 or 16 CPUs? These are important questions.

************** IMPORTANT CONCEPTS **************

o Pipelined vector units offer the potential of a factor of 20 improvement in computational performance over scalar architecture. Parallel CPUs, say on the order of 16, offer a factor of 16 improvement. Combined, the two architectural approaches offer the potential of 320 times conventional performance.

o This potential is only realized through superior computational strategies utilizing long vectors and low overhead task synchronization schemes.

o Parallelism can be applied at several levels

- job stream
- job step
- within the job step

o There are two approaches being used today:

- top-down (asynchronous MIMD)
- bottom-up (inner-loop parallelization, SIMD)

o Over the long term, top-down parallelism holds the most potential. Over the near term, a combination of both approaches seems likely.

****

3.5 THE ART OF BENCHMARKING SUPERCOMPUTERS

Throughout this, and the previous chapters, a great deal of discussion has been f ·sed on the differences, both in hardware and performance, among the supercomputers considered. A methodology for using these computers in a fashion that promotes efficiency and some degree of portability was also discussed. Much of the material presented has a bearing on another aspect of computing, performance prediction. Many organizations face the task of selecting a supercomputer in order to replace aging computing equipment. A very common practice over the years to aid in this process has been the "computer benchmark". This is a simple process. Select important programs or job streams and "test" them on the candidate computers and compare results. The fact is this process simply doesn't work in today's environment. Conventional benchmarking methods often lead to spurious results. In this section we will discuss benchmarking as both a topic of interest, and as a method to summarize much of the material presented in Chapters 2 and 3.

3.5.1 Background

Computer performance evaluation has long been a challenging field. The problem of predicting a computer's performance, either as an isolated system or relative to other systems, is multifaceted. Selecting a benchmark suite is probably the most difficult task anyone could face. This discussion is not about selecting benchmarks that characterize job streams, but rather a discussion of how to design the benchmark to effectively interpret the results.

The relation between the subtle architectural features of modern supercomputers and the computational structures of application programs has been shown to be very critical to performance. This relation is also very sensitive. A small architectural variation can lead to a larger performance variation. Using mathematical terminology, the benchmark process is can be termed an "unstable" process. That is, small variations in the input parameters (such as machine characteristics, compiler performance, library kernel software, and even application program input parameters) can result in very large changes in performance. Consequently, even a simple matter of obtaining a performance benchmark of a single application program can be very complex. Some benchmarkers attempt to hold all variables constant and require that applications be benchmarked without variation from system to system. As a result, the performance penalty is as much as a factor of 8 or 10 slower than the true potential of the computer involved. (Recall Figure 3-25.) What then is learned by taking a program that will essentially be running at one eighth its true potential on a vector machine, and using it as a performance measurement tool? What is learned by benchmarking an inappropriate scalar program on a vector computer? Clearly, we reject this approach.

An alternate approach, to assessing computer performance quickly and "fairly", is the through the use of kernel benchmarks such as the LNLL Fortran Kernels [3-15], or the Argonne Linear Equation Kernels [3-1]. In fact, there are a half a dozen, or so, kernel benchmarks commonly used in the government and private sectors for evaluating computer performance. Years ago, when computer architectures were more monolithic (and of a scalar character) the benchmark process was more stable, particularly at the kernel level. (In those days, how may millions of instructions per second (MIPs) a CPU could process was a valid rating of computational performance. Of course, MIPS have very little application to vector hardware.) Another complication is that various machines are better suited to some kernel computations than others. Consequently, algorithm designers employ different algorithmic strategies on different machines, even when computing similar higher level computations. Thus, performance potential cannot always be adequately tested by a single untouched "Fortran source". There are many important machine dependent optimizations that compilers cannot handle.

Recently, the IEEE Subcommittee on Supercomputing and a National Academy of Science Committee independently decided to investigate the benchmark process. The hope was to recommend a method or set of benchmark tools. After some months several approaches seemed to show promise, but the consensus was that this topic is so formidable that it deserves a solid research investigation. In the discussion that follows some of the problems in interpretation of benchmark results are examined.

3.5.2 The Impact of Hardware on Benchmarking

Table 3-9, below, indicates that architectures, even within one company's product line (in this case CRI), can have a great deal of variation. These architectural differences can have a tremendous impact on performance, and require compensating changes by the system, compiler and/or user, if acceptable performance is to be obtained. To appreciate this fact, let's examine the CRAY-1 and X-MP data in the table. The CRAY-1 has one path-to-memory, allows for only regularly stored vectors, and has more difficulty chaining operations than the X-MP. On the other hand the X-MP has 3- paths to memory, liberal chaining, allows for random vector operations via the added gather/scatter instructions. The CRAY-1 has a 12.5 nanosecond cycle time and the X-MP has 9.5 nanosecond cycle time (recent models are 8.5). If these were purely scalar machines, the performance difference would be 25% improvement on the X-MP.

Architectural Differences in the Cray Product Line

FEATURES	CRAY-1	X-MP(OLD)	X-MP(NEW)	CRAY-2
CLOCK in nanosec.	12.5	9.5	9.5 (8.5)	4.1
CPU's	1	4	4	4
PATHS/MEM	1	3	3	1
VECTOR STORAGE	REGULAR	REGULAR	RANDOM	RANDOM**
CHAINING	SLOT	FULL	FULL	NONE
COLLAPSED SUM	YES	NO	NO	NO
MEM/WAIT CYCLES	11-13	14	14	55
MEMORY (MW)	4	4	4 (16)	268

**Note: even though the Cray-2 supports random vector storage the underlying gather/scatter operations are 4 times slower than the regular vector rate. This is not the case on the X-MP

TABLE 3-9

Recall the 7 computational kernels examined earlier.

1. SAXPY - scalar times a vector plus a vector
2. CAXPY - complex version
3. SAXPYS - sparse vector SAXPY
4. SDOT - dot product
5. ISAMAX - finding the maximum component of vector
6. Matrix Multiply
7. Complex FFT

All these kernels were coded in near optimal CAL on the CRAY-1 and X-MP. The results in MFLOPS for large values of vector length are given below. (Also refer to Figure 3-22.)

KERNEL	CRAY-1	X-MP	X-MP (CFT 1.13)
1	50	180	130
2	100	175	110
3	4	87	4
4	75	190	200
5	28	35	4
6	160	195	100
7	95	170	55

The above figures indicate that not only does the kernel behavior radically differ between computers, but that reliance on Fortran could obviate the architectural gains offered in the X-MP design.

In summary, the selection and use of proper computational kernels can make an impressive difference in the performance of application programs. Furthermore, the kernels themselves may require optimization that the compiler alone cannot be depended upon to provide.

3.5.3 Interpretation of Kernel Benchmark Data

Probably the most quoted benchmark test kernels are the Argonne Benchmarks and the LNLL "Livermore" Kernels (referenced earlier). The providers of these benchmark kernels are experts in their fields and fully appreciate the value and use of the data generated from these benchmarks. In the case of the LNLL kernels, there is a solid understanding of the relation between each individual kernel's performance and important application programs often used at LNLL. However, the reporting and use of these benchmarks by manufacturers and "third party" performance evaluators has resulted in a great deal of misinformation. (This is not the responsibility of the kernel developers, however.) It is appropriate to examine the nature of the misuse of this data. In probing the weaknesses in computer evaluation techniques, one can begin to formulate strategies to improve the benchmark process as well as develop strategies to better interpret published results.

The Argonne benchmark is based on one simple, yet often used, application, the solution of linear systems. It is based on the subroutines found in LINPACK [7], one of the most commonly used public domain software libraries ever written. The first problem one encounters with kernel benchmarks, is the compiler. Does one wish to benchmark the computer hardware or the combination of the hardware and systems software, such as the compiler? In fact, compilers are so critical in this area that the original distributions of LINPACK had the inner loops "unrolled" in order to provide scalar compilers better opportunities to optimize. This "unrolling" is actually a negative factor on vector computers, so the Argonne benchmark often talks of "rolled" loops (which are the natural Fortran implementation). In addition, the Argonne benchmark allows for manufacturers to report assembler "coded" inner loops. The difference between coded and rolled performance can be substantial. For example, according to reference [3-1], the coded X-MP rates are 1.8 times faster than the CFT rates for systems of size 100. Another limitation of the CFT compiler is the fact that, on these kernels, performance can be enhanced further by optimization of the first two levels of nested inner-loops. This often requires a significantly different sequence of operations. The Argonne report calls this the matrix/vector (MV coded) approach when implemented in assembler.

With this as background one can observe many anomalies in reported results (not by the Argonne report itself, but from misleading use of the data by others). For example, in Figure 3-28 the results are lifted directly from reference [3-1].

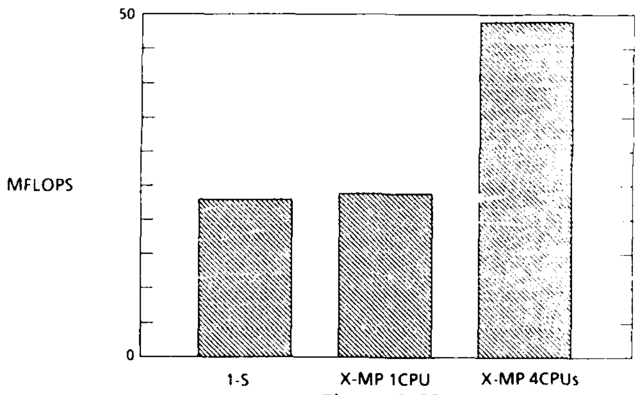


Figure 3-28
Inappropriate Use of Benchmark Data

The information as to whether the results are coded or rolled, is omitted. This is quite often the kind of information presented by manufacturers comparing different computer companies. The figure only contains CRI computers. At face value one would conclude the CRAY-1 and CRAY X-MP are of similar performance for a single CPU. The "dishonesty" in our presentation of Figure 3-28 is that rolled and coded results are being mixed. A rolled CRAY X-MP is not much better than a carefully coded CRAY-1, but this is an apple and an orange comparison! Further one might conclude that 4 CPUs only afford a factor of two improvement. While this is a true for this problem, it may be quite misleading, as will be discussed.

Figure 3-28 gives the more "honest" presentation of the data, plus the larger problem size of 300 is used with MV coded. This is a more honest test of the hardware because the compiler deficiencies are eliminated by allowing more optimal coding. The problem size more readily reflects the asymptotic rates attainable with larger vectors. A completely different set of conclusions can be observed. The X-MP is two times faster than the CRAY-1S, and the 4 CPU version (which runs at 480 MFLOPS) is almost three times faster than the single CPU X-MP. This information is, in fact, in an appendix of the Argonne report. At this point the reader of reference [3-1] might feel he has got a handle on the performance of dense linear equation algorithms on the CRAY line. However, reading still further in the appendix one finds another table, for problem size 1000 -- with no constraints on the test. In this case the 4-CPU X-MP runs at 713 MFLOPS, a 1.5 times faster rate than the rate achieved in problem size 300.

The larger problem size ameliorates (through larger granularity of tasks) the overhead of the multitasking tools used. Whether macrotasking or microtasking was used is not clearly stated. Thus, it is not clear as to whether the 713 figure is optimal.

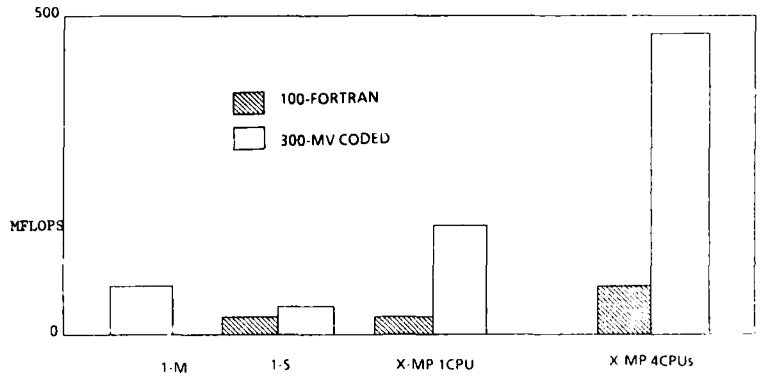


Figure 3-29
Appropriate Use of Argonne Benchmark Data

What then is the proper data to use -- problem size 100, 300, or 1000, coded or rolled (or unrolled as distributed in LINPACK), multitasked or single CPU? The fact is there are no definitive answers without a thorough knowledge of the ultimate use of the data. Rating a computer's performance on one algorithm, no matter how well understood or implemented, is simply one data point on an infinite spectrum of useful computational approaches in scientific computation. (Also note, in a similar vein, when observing third party quotations of the Argonne data, one should be alert to anomalies in data reporting. Is the presenter comparing computer timings from the same Argonne report, latest compilers, coded or not, same problem size, etc.) The final question is, obviously, does the Argonne benchmark have any bearing on applications to be ultimately used by those seeking an evaluation.

In the quest for reasonable benchmark data it is logical to broaden the kernel set to include other algorithms of importance to scientific computation. The LNLL benchmark kernels are an example. Twenty-four kernel computations are provided. They are executed with various loop sizes. Statistical information such as range, mean, median, average rate, and standard deviation are reported for each loop. A facility to obtain a weighted mean (arithmetic and geometric) for all loops is provided as well. The developers have also gone further at LNLL by providing a carefully instrumented weighted average that achieves the same MFLOP averages as observed in the Lab's job stream. The value of this however, eludes this author. On the other hand, and most importantly, in discussions with those who support these loops, one is quite impressed by the apparent "connection" that is understood between a particular loop and the type of application it represents. If one can make such a "connection," then the loop performance can be translated into truly meaningful information. This in fact, is the basis of a methodology for benchmarking that will be discussed in the next section. First, it is worthy to discuss some caveats in using the Livermore Kernels for comparing machines.

First, the Livermore Kernels are Fortran benchmarks. Therefore, the results are not a test of the machine, but a test of the machine and compiler. This is neither good nor bad, simply one of the facts. Another caution is related to loop titles and possible inferences drawn from these titles. For example, let's assume that an important application uses algorithmic approaches that require frequent solution of tri-diagonal equations. One may be inclined to carefully look at the data for Loop 5, called "Tri-diagonal Elimination". In fact, there are important applications resulting from partial differential equations that give rise to hundreds or thousands of such systems to be solved in a single job. However, it is quite often the case that these systems are independent and can be solved simultaneously. The inner loops of such a formulation have little in common with the recurrence loop used in "Loop-5''. Very incorrect conclusions could be drawn by the uninitiated. This, of course, is not a condemnation of the Livermore Loops, but of their potential misuse. Another example of a potential problem of Fortran-only benchmarking can be observed by Loop-24, "Find location of first minimum in array" given below:

LOOP 24: FIND MINIMUM OF ARRAY

```
      X(N/2)  =  -1.0E+10
      DO  24  L  = 1,LP
          M = 1
      DO  24  K  =  2,N
          IF (X(K).LT.X(M)) M = K
   24 CONTINUE
```

Some high performance computers actually have machine instructions that can implement this loop in one or two vector instructions. The CRAY X-MP has the ability to perform this loop in very concise set of vector instructions. The problem here is that many compilers would find optimization of this loop somewhat difficult as written (even with compiler directives). The same types of questions re-enter the discussion. What is it one wants to benchmark, the compiler, or the hardware potential?

In an effort to assess "dusty deck" or "average" code, the Livermore Kernels include Loop 15, "Casual Fortran.'' The code here is taken out of some "typical" program. What does its performance indicate? Whose casual Fortran is it? Even if Loop 15 is indicative of every day coding, is that what a benchmark is about? Perhaps, but its importance is at best, only relative to the user's purpose and concerns. How then is this loop to be included in statistics such as weighted averages.

In summary, kernel benchmarks are indeed important. In fact, as will be asserted, they are a good start to ascertaining performance potential, if used properly. However, cold benchmarking of kernels on hardware of different types without a clear understanding of the architectural sensitivities and the connection to particular applications is, at best, dangerous!

3.5.4 A Benchmarking Methodology

In designing applications programs on modern computing machines, there are often stages of modeling and implementation that result in a "running code." Most of what performance analysis (and hence benchmarking) is about, is assessing the performance of the "running code" as it exists in its most stable production-grade form. As mentioned earlier, what really is important, however, is to measure the potential performance of the scientific problem as it could be implemented on new hardware. Somehow in dealing with "vector" computers, this fact has been hard for many to accept. Consequently, some organizations will make purchase decisions based on the benchmark process where the ground rule is "hands-off the source," i.e., do not attempt any optimization other than what the compiler can do. This approach is fortunately losing favor, and will no doubt fall by the way completely as we move into the age of parallel (and vector) CPUs. Compilers or systems that can effectively automate the process of conversion and optimization in a parallel environment are many years away. Understanding the steps in program design can perhaps help in understanding possible approaches to benchmarking, both the program as it "is" and gain insights into what it "could be" on a new system (real or in design).

An approach to benchmarking computers, that isolates some of the anomalies discussed thus far, can now be addressed. First, the key to the process is a thorough understanding of the application program being benchmarked. The stages of its development including important computational kernels and alternate computational models must be understood. The "as-is" benchmark will provide an important data point, but cannot be interpreted without individual kernel benchmarks. It is to be emphasized, that providing benchmarking kernels in Fortran and in optimized assembler if required is the easy job -- the difficult part of benchmarking is establishing the performance "connection" between kernels and application performance.

Often benchmarks offered by organizations planning to purchase computer equipment focus on the job stream or load. Characterizing the load by a limited number of jobs is a problem at least as difficult as the ones expressed so far. It is particularly perplexing in the supercomputing arena. For example, the biggest most resource consuming job in a company's scientific load, may be a job that pushes computation and I/O to the limits. Perhaps this is due to the fact that it requires more real memory than available in the system. In the supercomputer environment such a program may become one of the minor or even discarded programs owing to the possibilities offered by the new capability. With that rather difficult caveat to overcome, following steps in benchmarking are suggested:

1. Identify the spectrum of applications important to load.

2. In each application identify computational "hot spots" and I/O bottlenecks

3. Examine alternate potentially efficient computational approaches at model, implementation, algorithm, and kernel levels.

4. Establish timing mappings from important kernels to important algorithms to application performance.

The above steps are the hard part. The benchmark step itself is easy. Run the applications "as-is", followed by individual tests of important algorithms and kernels for both existing and alternate algorithmic approaches. In addition, where a computational kernel is critical to performance, optimize it by whatever means necessary (i.e. don't rely solely on compilers for optimal performance.) This is important even if the ultimate goal is to have all models running from Fortran. This will give one a better picture of potential performance.

The expense of this approach can be considerable. However, if done correctly, the benchmark will dictate a conversion strategy and new approaches to computational strategies on the target machine. The truly expensive part is the characterization of load, algorithm, and kernel hot spots, and the mapping connecting these levels of complexity to the application. This analysis is only required once for all computers, if done well. The result is a benchmark methodology directly tied to the organization's load. In times of pressing resources or when a quick evaluation of a computer is required (perhaps for screening purposes), the benchmark kernels alone can be used to give meaningful benchmark results by relating back to timing models for the existing load AND alternate approaches. Thus, both an estimate of existing load and potential performance can be characterized from a rather simple test. With this kind of load characterization, the actual full application benchmark will give the least amount of insight into the performance of the new hardware. In fact, what has just been described is not far from the use that the functional fathers of the Livermore kernels make of their kernel benchmarks, for they have a "feel" or "connection" between the kernels and important applications that characterize their load. This may not be true for all the Lab scientists, and is certainly not true for most of the rest of the world who are sometimes influenced by kernel benchmarks of this kind.

AN EXAMPLE

A true awareness of critical parts of an application program can only be attained by an intimate understanding of the program's use and the flow of the program's input, computation, and output. It is very difficult for someone who sees a program listing for the first time (even with good accompanying documentation) to be able to gain this type of awareness. However, once a clear documentation of the program's computational "hot spots" are identified, optimal benchmarks can be obtained. An example of a program profile can illustrate what is required. Consider the following timing profile of a Power Flow application used in the electric power industry.

POWER FLOW

FUNCTION	CPU PERCENTAGE
MATRIX SETUP/INPUT	7 %
MERGE CASE OR REDUCTION	6 %
SOLUTION (SPARSE MATRIX)	55 %
OUTPUT ANALYSIS	32 %

TABLE 3-10

What the profile above does not reveal, is that roughly 90 percent of the code listing is outside the solution phase. The computational kernels that perform roughly 55% of the computation are rather small. In fact, if one were to benchmark the sparse matrix kernel alone, a tremendous amount would be learned about the performance of the entire code. Another thing that is not revealed by the profile, which is critical to a benchmark process, is that the code can be used much differently. As it stands, even if one reduced the computationally oriented section (the solution step) to zero percent, the code improvement would be only reduced by a factor a little better than two. However, if one knew that the code is really used iteratively with the user in the loop to project effects of new power generation placement or the effects of transmission line malfunction, another program could be conceived. One in which the same solution step is run many times against automatic changes in the input. This would eliminate the need to setup all the input, merge cases, and analyze all the output. In this more sophisticated model, the solution phase is done repeatedly and can account for 90 % of the job time. Thus, by benchmarking the sparse matrix kernel, one can infer performance of a program yet to be designed, and predict real long term cost benefits from high performance hardware upgrade. In effect, one can benchmark today's program and infer performance about tomorrow's load. It is against this model, of in-depth understanding of a particular application, that benchmark studies should be developed. As difficult and challenging as this may be, it is well worth the effort. In fact, it has been proposed that industry sectors cooperate to achieve meaningful industry benchmark methods. For example, aerospace or petrochemical benchmark suites could be conceived that would help both the respective industries and hardware vendors. These suites could benefit greatly by mixing both applications and important underlying kernels together. At the very least, the important kernels with documentations as to how they interact with applications, can provide an important benchmarking tool.

In summary, benchmarking supercomputers is a difficult and often misunderstood process by many potential users. The method or approach outlined above is not as simple as it appears. Characterizing an application's performance from underlying kernels is not simple. At the outset it was mentioned that the author believes benchmarking supercomputers is, indeed, a research area. Nevertheless, increased understanding of benchmark issues can lead to a more scientific approach to the process. Important issues in the benchmark process include a firm understanding of what information is being sought. Does the benchmark adequately predict existing load performance or future load performance? Does the benchmark indicate the potential for new approaches to computational problems or even the feasibility of new physical models? Finally, it is recognized that kernel benchmarks such as those mentioned are important tools if properly used, but if used without connection to the desired applications their performance can bear little relation to load performance. This is well recognized by those who provide kernel benchmark data and must be understood by the community that interprets these data.

************ **IMPORTANT CONCEPTS** ************

o Benchmarking is an unstable process. It is sensitive to architecture, algorithms, and sometimes test case data.

o The selection and use of optimal kernels within an application can improve benchmark results dramatically. This also separates the negative effects of the compiler.

o The commonly used kernel benchmarks can be misused quite easily. The usefulness of kernel benchmarks depends on a sound understanding of the relationship between the computational kernels and actual application programs. Untouched benchmarks of applications themselves, can be just as misleading.

o A Benchmark Methodology:

1. Identify the spectrum of applications important to load.
2. In each application identify computational "hot spots" and I/O bottlenecks.
3. Examine alternate potentially efficient computational approaches at model, implementation, algorithm, and kernel levels.
4. Establish timing mappings from important kernels to important algorithms to application performance.

****

3.6 REFERENCES

[3-1] Jack Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Technical Memo. No. 23, Mathematics and Computer Science Div., Feb. 27, 1986.

[3-2] K. Neves, "Mathematical Libraries for Vector Computers," Computer Physics Communications, Vol. 26, 1982, 8 pages.

[3-3] R. Hockney, "Performance of Parallel Computers," Proceedings, NATO Advanced Research Workshop on High-Speed Computation, Juelich, W. Germany, June, 1983.

[3-4] K. Fong and T. Jordan, "Some Linear Algebraic Algorithms and their Performance on the CRAY-1," Proceedings of the Symposium on High Speed Computer and Algorithm Organization, Academic Press, New York, 1977, pp. 313-316.

[3-5] B. Dembart and K. Neves, "Sparse Triangular Factorization on Vector computers," Exploring Applications of Parallel Processing to Power Systems Problems EPRI EL-566-QR, October 1977, pp.57-103.

[3-6] I. Duff, A. Erisman, and J. Reid, Direct Methods for Sparse Matrices, Clarendon Press, Oxford, 1986.

[3-7] Sources and Development of Mathematical Software, W. Cowell, Ed., Prentice-Hall Series in Computational Math., 1984.

[3-8] A. Erisman, K. Neves, I. Philips, "The Boeing Mathematical Libarary", IBID.

[3-9] K. Neves, "The Impact of CRAYS's Changing Architectures on Scientific Computation," Proceedings, Cray User Group, Stockholm, April 1985.

[3-10] P. Fox, "The PORT Mathematical Subroutine Library," Sources and Development of Mathematical Software, W. Cowell, Ed., Prentice-Hall Series in Computational Math., 1984.

[3-11] FACOM VP System Performance Data, Fujitsu Ltd, private communication, 1985.

[3-12] R. Grimes, J. Lewis, H. Simon, "Eigenvalue Problems and Algorithms in Structural Engineering," Large Scale Eigenvalue Problems, . J. Cullum and R. Willoughby, eds., Elsevier, North-Holland, 1986, pp. 81-93.

[3-13] S. Chen, "Overview of the CRAY X-MP," Presented at Cray Research Visit, 1984.

[3-14] Vectorpak Manual, (A product of Boeing Computer S..vices, P.O. Box 24346, Seattle, WA, 98124.)

[3-15] F. H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," UCRL-53745, LLNL, U. of CA, Livermore, CA., December, 1986.

[3-16] J. Dongarra, J. Bunch, C. Moler, and G. Stewart, LINPACK Users, Guide, SIAM Publications, Philadelphia, 1979.

[5-17] K. W. Neves, "Trends in Supercomputer Architecture," Proceedings, Ninth Conference on Electronic Computation, ASCE, Feb. 1985.

[3-18] J. Lewis and H. Simon, "Impact of Hardware Gather/Scatter on Sparse Gaussian Elimination," BCS Technical Report, ETA-TR-33 (PO Box 24346, MS 71-20, Seattle, WA 98124) June, 1986.

CHAPTER 4: VECTORIZATION OF FORTRAN PROGRAMS AT DO-LOOP LEVEL (Prof. Dr. Gentzsch)

4.0 INTRODUCTION

In the previous chapters the impact of the computer hardware on algorithm performance was discussed from a general point of view. It was made clear that the most effective use of supercomputers comes from sound understanding of computer architecture and an appropriate computational strategy. In this and subsequent chapters, the focus will be on fluid dynamic applications. Today, and for some years to come, the major interface between the scientist and the computer will continue to be the Fortran language. We will present in this chapter some common techniques useful in dealing with code optimization within the critical "loops" of Fortran, and in Chapter 5 the important consequences of restructuring the linear algebra algorithms common in CFD applications. Finally, in Chapter 6, major applications will be reviewed from a computational strategy perspective giving a broad-based approach utilizing proper Fortran, proper algorithms, and awareness of architectural dependencies. Here we shall restrict ourselves to discussing some helpful coding techniques, which are easy to understand and to implement. However, it should be emphasized, that the selection of an appropriate algorithm is generally far more important than fancy programming techniques (see chapter 5).

4.1 IMPLEMENTATION OF SERIAL PROGRAMS ON VECTOR COMPUTERS

The production codes in any field of computational physics are usually highly complex. The vectorization of many thousands of statements will naturally frighten every user. With the following strategy, however, an efficient vectorization of only parts of the program substantially improves efficiency, while not being particularly arduous [8]:

STEP 1: Generation of a histogram showing the amount of CPU-time for different sections of the program. Table 4.1 shows a Flow Trace for a multigrid solution of the Helmholtz equation on a CRAY-1. As is demonstrated, about 86 percent (related to 0.86 in the histogram) of the total CPU time is spent in the subroutine RELAX solving the systems of linear equations with different mesh sizes. This is typical for most production codes arising in the numerical treatment of differential equations. A more detailed code analysis is possible on most of the vector computers. For example, for the CRAY X-MP, static analysis tools such as FTREF and VMARK provide the user with information on the references to Fortran variables and vectorization potential at compilation time. During program execution, dynamic analysis software such as FLOWTRACE and SPY supply the user with timing information (from the hardware performance monitor).

Table 4.1: Simplified flow trace for a multigrid code on a CRAY.

	ROUTINE	TIME	%	CALLED	
1	FNH1R1	0.000395	0.00	1	
2	GRDFN	0.000061	0.00	21	
3	PUTZ	0.003958	0.04	61	
4	KEY	0.017824	0.17	1540	
5	PUTI	0.094767	0.90	8	
6	F	0.028835	0.27	16129	
7	PUTB	0.000375	0.00	1	
8	G	0.000915	0.01	512	
9	C	0.038148	0.36	21343	
10	CNFIX	o.058612	0.56	10	
11	RELAX	9.056721	86.16	370	← vectorization
12	RSCAL	0.323413	3.08	60	
13	INTAD1	0.888053	8.45	60	
***	TOTAL	10.512078			
***	OVERHEAD	1.121270			

STEP 2: Hand-tailor the most time-consuming subroutines if auto-vectorization does not suffice. The CYBER and the CRAY compilers present, at the end of each subroutine, a complete list of the vectorized and non-vectorizable loops together with the reasons for non-vectorizability. A nonlinear recursion is evaluated in DO-loop 20 of the next example (part of the Thomas algorithm discussed in the next chapter) and the reason for non-vectorization is given for a CRAY:

```
10.   O(2) = 4.
11.   DO 20  I = 3, M
12.   EL(I)=1./O(I-1)
13.   O(I) =4. -EL(I)
14. 2C CONTINUE
```

```
AT SEQUENCE NUMBER - 13.
PRNAME   LJAC     COMMENT- DEPENDENCY  INVOLVING  ARRAY  "O"   IN
                  SEQUENCE NUMBER 12
```

STEP 3: In the case of a highly serial algorithm involving, for example, linear and non-linear recurrences, STEP 2 is not successful and a complete restructuring of the algorithm is necessary. We will discuss this step in chapters 5 and 6 for many examples.

4.2 VECTORIZATION EXAMPLES FOR CRAY COMPUTERS

The CRAY computers are register-to-register vector computers as described in Chapter 2. For this type of computers we present some helpful restructuring techniques and their implementation in computer programs such as (see [3], [4], [5], [8]-[11], for more detail)

- Putting DO-loops into subroutines or functions and vice versa
- Using few loops with long code blocks in preference to many short code loops
- Using long loops inside short loops rather than vice versa
- Special subroutines for linear recurrences
- Partial vectorization of irregular addressing
- Removing IF statements
- Manipulating operations so that they occur in an order that increases chaining.

This is by far not a complete list of loop vectorizations but gives the reader a first impression of code modifications to obtain adequate vector structures.

The FLOW TRACE option is used first to obtain a complete list of the subroutine calling tree and the time spent in each routine. One then starts vectorizing the most time consuming parts of the program.

As there are few FORTRAN extensions provided for the CRAY FORTRAN compiler, most of the problems treated in this section deal with the restructuring of a sequence of standard FORTRAN statements. For example, subroutine and function calls within DO loops depending on the loop indices prevent the compiler vectorizing. The following sequence might have arisen in a program solving a finite difference equation (M1=M-1):

```
   DO    1  J = 2, M1
   DO    1  I = 2, M1
   PM = VELOC  (I,J)
   PL = VELOC  (I-1,J)
   PR = VELOC  (I+1,J)
   CALL RELAX (PM, PL, PR)
   VELOC (I,J) = SQ2 (PM)
 1 CONTINUE
       .
       .
       .
   SUBROUTINE RELAX (PM, PL, PR)
   COMMON OM, HH
   PM = (1.-OM) * PM + 0.5 * OM * (HH + PL + PR)
   RETURN
   END

   FUNCTION SQ2 (P)
   DATA ALPHA /.../
   SQ2 = ALPHA * SQRT (P)
   RETURN
   END
```

The innermost DO loop will not vectorize owing to the subroutine call and the call to a function not recognized by the compiler. Putting the loop inside the subroutine leads to

```
      .
      .
      .
   CALL RELAXV (VELOC)
   CALL SQ2V (VELOC)
      .
      .
   SUBROUTINE RELAXV (U)
   DIMENSION U (100, 100)
   COMMON OM, HH, M1
   OM1 = 1.-OM
   OM2 = 0.5 * OM
   DO    1  I = 2, M1
   DO    1  J = 2, M1
   U(I,J) = OM1 * U(I,J) + OM2 * (HH + U(I-1,J) + U(I+1,J))
 1 CONTINUE
   RETURN
   END
   SUBROUTINE SQ2V (V)
   DIMENSION V(100, 100)
   COMMON OM, HH, M1
   DATA ALPHA /.../

   DO    1  J = 2, M1
   DO    1  I = 2, M1
   V(I,J) = ALPHA * SQRT (V(I,J))
 1 CONTINUE
   RETURN
   END
```

Both subroutines will now vectorize. But in this example it would be better to put the subroutines inside the loop to increase the arithmetic in the inner loop:

```
     .
     .
     .
   DO    1  I = 2, M1
   DO    1  J = 2, M1
   VELOC (I,J)= OM1 * VELOC (I,J) + OM2 * (HH + VELOC (I-1,J) +
                VELOC (I+1,J))
   VELOC (I,J)= ALPHA * SQRT (VELOC (I,J))
 1 CONTINUE
     .
     .
     .
```

In this example, aside from the improved vectorizability, the program also achieves increased transparency. Notice that the I-loop is the outer loop because of the dependence corresponding to this index.

The above instruction sequence is also an instructive example in the use of as few loops as possible, containing long code blocks, instead of many short-code vectorizable blocks. Consider the following sequence:

```
   CALL  VADD (A,B,C,N)
   CALL  VMULT(C,A,E,N)
   CALL  VADD (E,B,A,N)
```

Here one uses the vector subroutines VADD and VMULT. This version vectorizes, but the expanded combination

```
   DO  1  I = 1,N
   A(I) = (A(I) + B(I)) * A(I) + B(I)
 1 CONTINUE
```

is significantly faster then the series of calls. The sum A+B and the product (A+B)*A do not have to be stored, but can be kept in a register and A does not have to be fetched a second time. This is also an example of the manipulation of operations in order to increase chaining. Consider for example

```
   DO  1  I = 1,1000
   DO  1  J = 1,5
   A(I,J) = (A(I,J) + B(I,J)) * A(I,J) + B(I,J)
 1 CONTINUE
```

Since only the innermost DO loops are vectorized, the calculation with vectors of length 5 leads to a performance rate similar to that of scalar performance. Reversing the order of the I and J loops would lead to an improvement factor of more than 10 over the original code depending on the computer.

One of the most difficult problems on vector computers is the vectorization of linear and non-linear recurrences. As this is more a question of algorithm, we shall return to it again later. For the moment however, we shall restrict ourselves to the implementation of single linear recurrences on the CRAY.

A linear recurrence uses the result of a previous pass through the loop as an operand for subsequent passes, and this prevents vectorization. An example of a first-order, linear recurrence is

```
   S(1) = A(1)
   DO   1   I = 1,N-1
   S(I+1) = -B(I) * S(I) + A(I+1)
 1 CONTINUE
```

A second-order, linear recurrence may be of the form

```
   S(1) = A(1)
   S(2) = A(2)
   DO   1   I = 1,N-2
   S(I+2) = B(I) * S(I+1) + A(I+2) * S(I)
 1 CONTINUE
```

In these cases straightforward vectorization is impossible. Therefore, CFT offers special subroutines which run with optimum efficiency on the CRAY, and which solve first-order, and some second-order, linear recurrences. The subroutine

```
   FOLR (N,A,INCA,B,INCB)
```

for example solves the above mentioned first-order linear recurrence. Here INCA and INCB are the skip distances between the elements of the vectors A and B, respectively. N is the length of the recurrence. The output overwrites the input vector B. On the X-MP with the CFT77 compiler this routine runs with more than 20 MFLOPS.

However, within more complex programs, vectorization of recurrences may still be straightforward (cf. section 5.3 for a more detailed discussion).

If the DO-loop is not truly recursive, as for example in

```
   DO   1  I = 200, 300
   A(I) = A(I-L)
 1 CONTINUE
```

and L has some positive integer values between 101 and 200, the easiest approach is to try directing the compiler to vectorize the loop and see if the answers remain the same. The compiler directive

```
   CDIR$ IVDEP
```

placed immediately in front of the DO-loop to be vectorized causes the computations to be performed in vector mode, provided the loop contains no CALL or IF statements.

Another example of fictitious recursions often arises in problems with red-black and zebra-line structures (cf. sections 5.4 and 5.5) which can easily be vectorized by applying the same compiler directive:

```
      DO   40   J = 2,M1,2
CDIR$ IVDEP
      DO   40   I = 2,M1,2
      U(I,J) = 0.25 * (U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1))
   40 CONTINUE
```

For fixed J, the even subscripted values of U on the left hand side depend only on the odd subscripted ones on the right hand side, and the directive is appropriate.

In many applications no contiguous data structure is present. In the Monte-Carlo method (see section 6.10) we have to deal with randomly distributed data, while in three-dimensional problems it is necessary to gather and scatter two- and one-dimensional substructures. In FORTRAN this problem is expressed by subscripts as in the following example

```
   DO   1  I = 1, 100
   J = INDEX (I)
   A(I) = B(J) + ...
 1 CONTINUE
```

For some vector computers without hardware gather/scatter this loop can partly be vectorized by using a temporary array to first gather the irregularly distributed elements into a contiguous vector:

```
   DO   1  I = 1, 100
   J = INDEX (I)
 1 TEMP(I) = B(J)
   DO   2  I = 1, 100
 2 A(I) = TEMP (I) + ...
```

For problems with irregular addressing, gather and scatter subroutines are available. The above example then simply reads as follows (e.g. for a CRAY-1)

```
   CALL   GATHER (100, TEMP, B, INDEX)
   DO   2  I = 1, 100
 2 A(I) = TEMP(I) + ...
```

The gather subroutine uses the integer values stored in the elements of the array INDEX as indices to take the random elements of vector B and make them contiguous in the vector TEMP. For many of today's vector computer, however, hardware gather and scatter are available. Therefore a hand-tailoring of these loops is no longer necessary for these computers.

As a last problem we deal with removing IF statements from innermost loops. Vectorization of some loops containing IF's may be straightforward while others are difficult but not impossible, depending on the structure of the code. Intrinsic functions may help to overcome some of these difficulties. The following example

```
   DO    1   I = 1, 100
   IF(A(I).LT.0.) A(I) = 0.
 1 B(I) = SQRT (A(I)) + ...
```

is already vectorized by the compiler and transformed into

```
   DO    1   I = 1, 100
   A(I) = AMAX1 (A(I), 0.)
 1 B(I) = SQRT (A(I)) + ...
```

which selects the maximum value of the two elements A(I) and 0.

In the next example, however, the user has to employ the vector merge operation CVMGT to merge the results of different vector computations (for example for inner and boundary points of a two-dimensional domain):

```
   DO    1   I = 1, 100
   IF(A(I).LT.0.)  GOTO 2
   B(I) = A(I) + C(I)
   GOTO 1
 2 B(I) = A(I) * C(I)
 1 CONTINUE
```

which can be converted to

```
   DO    1   I = 1, 100
   B(I) = CVMGT(A(I) * C(I),A(I) + C(I), A(I).LT.0.)
 1 CONTINUE
```

Other vectorization aids are explained in more detail in [5]. Furthermore a list of all scientific application subprograms is available and a brief explanation may be found in the Library Reference Manuals of the companies.

4.3 VECTORIZATION EXAMPLES FOR THE IBM 3090VF

Some additional examples deal with restructuring of simple DO-loops with regard to the architecture of the IBM 3090 with Vector Feature (VF), which in one important point is very different compared to other vector computers: namely, the cache memory between main memory and vector registers. Although many loop constructs are automatically vectorized by the compiler, rearranging the loop contents with respect to

* Cache utilization
* minimizing load/store
* keeping data in registers

often improves efficiency remarkably. The examples, taken from [5],[6],[9]-[11], consist of different Fortran versions dealing with a special problem. Performance in MFLOPS depending on vector length N is shown and the results are interpreted, for the following cases:

AVOID TEMPORARY ARRAYS

```
      Case 1:              Case 2:              Case 3:

      DO 100 I=1.N         DO 200 I=1,N         DO 300 I=1,N
      T(I)=A(I+1)          Y=A(I+1)             A(I)=C(I)
      A(I)=C(I)            A(I)=C(I)            B(I)=A(I+1)
      B(I)=T(I)            B(I)=Y           300 CONTINUE
  100 CONTINUE         200 CONTINUE
```

Table 4.2: MFLOPS (without and with cache miss)

N	Case 1		Case 2		Case 3	
8	2.7	2.4	3.7	2.8	4.0	3.9
16	4.7	3.1	6.2	3.6	6.8	5.5
32	7.0	4.4	9.9	5.6	10.8	8.4
64	9.9	5.7	14.4	8.3	15.3	10.7
128	13.6	7.9	18.8	12.9	19.4	15.7
256	13.7	7.4	19.9	12.2	18.9	14.1
512	14.8	7.1	19.7	11.1	20.2	13.1
1024	15.6	7.6	20.2	12.2	20.2	14.7

Comment:

The compiler vectorizes all three cases. However the best speed-up is obtained for case 3 because it involves less storage than cases 1 and 2. Indeed, case 1 with four vectors involved is the slowest. The more vectors are involved, the more the influence of the cache can be seen in cases, where the cache has to be emptied and refilled before the operations can start, which is shown in the second columns. Again, case 3 with the fewest number of vectors performs best.

As can be seen from the table, the use of the temporary variable Y in case 2 is nearly as fast as case 3. The reason is that the scalar variable T conceptually is expanded into a temporary array which appears only in a vector register and never in storage.

AVOID SCALAR VARIABLES WHICH ARE CALCULATED BEFORE THE EXECUTION OF THE CONTAINING LOOP

```
Case 1:                Case 2:

    R =0.                  R(1)=0.
    DO 100 I=2,N           DO 200 I=2,N
    S=A(I)*B(I)            R(I)=A(I)*B(I)
    C(I)=S+R               C(I)=R(I)+R(I-1)
    R=S                200 CONTINUE
100 CONTINUE
```

Table 4.3: MFLOPS for case 1 and 2

N	Case 1	Case 2
8	6.9	1.8
16	6.5	3.1
32	6.3	5.6
64	6.2	6.3
128	6.1	9.1
256	6.0	10.1
512	5.9	12.3
1024	5.9	13.6

Comment:

In case 1, because of presetting the scalar R to zero, R is not expanded automatically into a temporary array. So, the first loop does not vectorize. The vector report message tells that vectorization is not possible because of the scalar variable R which uses a value that is set before the execution of the containing loop. Case 2 is automatically vectorized.

AVOID IF-STATEMENTS IN LOOPS

Case 1:

```
    K=N/2
    DO 100 I=1,N
    IF(I.GE.K) C(I)=A(I)*B(I)
100 CONTINUE
```

Case 2:

```
    K=N/2
    DO 200 I=K,N
    C(I)=A(I)*B(I)
200 CONTINUE
```

Table 4.4: MFLOPS (without and with cache miss):

N	Case 1		Case 2	
	without	with	without	with
8	1.4	0.9	1.9	1.2
16	2.2	1.3	3.4	2.0
32	3.1	2.2	5.7	3.0
64	3.9	3.3	8.6	7.4
128	4.1	3.0	11.6	5.6
256	4.1	2.7	12.6	4.6
512	4.2	3.2	13.0	7.1

Comment:

Both cases are vectorized by the compiler. However, a logical mask has to be created under which computations in case 1 are performed. This results in a performance decreasing.

Another important point should be mentioned arising with very simple loops such as the multiplication of two vectors: Performance is increased remarkably using longer vectors. In small 2-dimensional arrays it is useful therefore to restructure the whole data into a long 1-dimensional vector. If this is too cumbersome, computation in scalar mode sometimes yields better performance.

UNROLL INNER LOOPS OF 2

Case 1:

```
    DO 100 I=1,N
    C(I)=0.
    DO 200 J=1,2
    C(I)=C(I)+A(J)*B(I,J)
200 CONTINUE
100 CONTINUE
```

Case 2:

```
    DO 200 I=1,N
    C(I)=A(1)*B(I,1)+A(2)*B(I,2)
200 CONTINUE
```

Table 4.5: MFLOPS for cases 1 and 2

N	Case 1	Case 2
8	2.4	6.0
16	3.1	10.4
32	2.7	17.1
64	2.2	25.7
128	2.5	34.7
256	2.8	36.2
512	2.9	38.2
1024	3.0	38.2

Comment:

Similar improvements (due to the COMPOUND operations) are to be expected for unrolling of inner loops of 4, 5, etc.

AVOID COMPONENT WITH CALL TO SUBROUTINES

Case 1:

```
    DO 100 I=1,N
    CALL ADD(A(I),B(I),C(I))
100 CONTINUE
```

Case 2:

```
    CALL ADD1(A,B,C,N)
```

Case 3:

```
    DO 300 I=1,N
    C(I)=A(I)*B(I)
300 CONTINUE
```

Table 4.6: MFLOPS for cases 1, 2 and 3

N	Case 1	Case 2	Case 3
8	0.5	1.4	2.1
16	0.5	2.5	3.6
32	0.5	4.3	5.9
64	0.5	6.8	8.8
128	0.5	9.9	11.8
256	0.5	11.2	12.4
512	0.5	12.4	12.6
1024	0.5	12.7	13.1

Comment:

The subroutine ADD is not analyzable within loop 100. Therefore this loop is not vectorized by the compiler. In case 2 the overhead of the subroutine call results in minor decrease in performance.

USE STATEMENT FUNCTIONS INSTEAD OF FUNCTIONS

Case 1:

```
    DO 100 I=1,N
    C(I)=FUNCT(A(I),B(I))
100 CONTINUE
```

Case 2:

```
    SFUNCT(R,S)=(R+S)**2
    .
    .
    .
    DO 200 I=1,N
    C(I)=SFUNCT(A(I),B(I))
200 CONTINUE
```

Table 4.7: MFLOPS for cases 1 and 2

N	Case 1	Case 2
8	0.6	1.8
16	0.6	3.2
32	0.5	5.1
64	0.5	7.3
128	0.5	9.5
256	0.5	9.8
512	0.5	9.9
1024	0.5	10.2

Comment:

Loop 100 is not vectorizable because the user function FUNCT is not analyzable. In case 2 the overhead of the statement function SFUNCT reduces the performance up to 30 percent.

USE VECTOR DIRECTIVES

Very often, necessary information for the optimal vectorization is not available at compile time. Therefore, vector directives may be used to support the compiler making appropriate decisions about vectorizations. At the beginning of the program unit the statement

PROCESS DIRECTIVE ('*VDIR:')

has to be specified with a user defined string '...'. The vector directives, available for most of the existing vector computers, are specified as comment lines directly above the loop to which they refer. For the IBM 3090 Vector Feature, e.g., the following directives are available:

C*VDIR: PREFER VECTOR
execute the following loop in vector mode

C*VDIR: PREFER SCALAR
execute the following loop in scalar mode

C*VDIR: ASSUME COUNT (n)
the iteration count of the following loop is the specified value

C*VDIR: IGNORE RECRDEPS
ignore potential dependences

C*VDIR: IGNORE EQUDEPS
ignore potential dependences between variables that are in an EQUIVALENCE relationship

Combinations of vector directives are possible. For most of the existing vector computers, similar vector directives are available.

4.4 SPECIAL VECTORIZATION HINTS

As a conclusion based on the previous examples a "checklist" for efficient vectorization especially at DO-loop level (step 2) can be set up which contains, among others, depending on the computer, the following items:

* make:
Innermost DO-loops most efficient

* avoid:
IF-statements
Subroutine and function calls
Irregular (nonlinear) addressing
Memory bank conflicts
Excessive paging
Complicated branching within a loop
Linear and nonlinear recursions

* use:
Many arithmetic operations within inner loops instead of many loops with few operations
Long vectors (i.e. large one-dimensional arrays with length equal to the number of grid points, instead of 2D arrays)
Unformatted I/O
Chaining possibilities and linked triads to achieve supervector speed (done by some compilers automatically)

* unroll:
short DO-loops

* separate:
Vectorizable from non-vectorizable parts

* switch
Inner and outer loops to have the longer index range on the inner loop and to suppress dependencies

In addition to this checklist some important rules concerning step 3, vectorization of numerical algorithms (see Chapter 5) with regard to vector computers are

* For the CYBER 205, change 2D and 3D arrays to 1D arrays

* Use diagonal storing for banded matrices

* Avoid gather/scatter on the CRAY-1S

* Vectorizable preconditioning in ICCG with approached inverse

* Assembler programming on the CRAY-1S and the CRAY-2 of the most time-consuming parts

* Use the Engineering and Scientific Library (ESSL) on the IBM VF

* Care has to be taken in implementing the boundary conditions on vector computers to maintain long vectors

* Solve many algebraic systems simultaneously (as in SLOR, ADI, POISSON, see chapter 5)

* Simplify algorithms using irregular addressing (such as pivoting in Gaussian elimination)

* Use red-black or zebra reordering in point or line relaxations (see chapter 5)

* Use simple elements (triangles in 2D, tetrahedra in 3D) and, if possible, hardware gather/scatter in finite element codes

* For unstructured grids use explicit schemes (see section 6.7)

* Improve the stability of explicit schemes which are very suitable for vector and parallel computers (see section 5.9).

For more details see [17].

4.5 SOME REMARKS ON MULTITASKING

While vector processing takes place at the level of innermost DO-loops, the division of a job into sub-tasks for multitasking should be as global as possible. Three strategies may be applied when simultaneously using two or more processors:

* Domain decomposition: Divide the computational domain into substructures to obtain independent problems in each subdomain to be solved on different processors

* Program decomposition: Split the computer code into smaller parts (e.g. subroutines with similar amount of CPU processing time) each of which is then solved on one processor.

* DO-loop decomposition: same as program decomposition, but on DO-loop level.

One efficient tool is dynamic load balancing to distribute the work equally among the different processors. Before multitasking, however, it is recommended to vectorize the computer program for the single processors.

For some special CFD applications, the concept of concurrently using multiple CPU`s for a single program has led to a speedup factor of up to 3.5 on a 4-processor computer, as compared to the vectorized, non-multitasked codes. A detailed analysis of speed-up is discussed in [14] and compared with the corresponding actual speed-up for real computations.

4.6 LITERATURE

[1] Besaw K.V.: Advanced Techniques for Vectorizing Dusty Decks. UNISYS Report 1986.

[2] Coleman H.B.: The Vectorizing Compiler for the UNISYS ISP. UNISYS Report 1986.

[3] CRAY-1 FORTRAN Reference Manual [CFT]. Pub. 2240009, CRAY Research, Minneapolis 1979.

[4] CRAY-1 Computer Systems FORTRAN (CFT) Reference Manual SN-0009, Revision I, CRAY Resarch Inc. 1982.

[5] CRAY Research Inc. Optimization Guide. CRAY Res. Publ. SN-0220, 1981.

[6] Dubrulle A.A., Scarborough, R.G.: Kolsky, H.G.: How to Write Good Vectorizable Fortran. IBM Techn. Report G 320- 3478, 1985.

[7] Gentzsch W.: A Survey of the New Vector Computers CRAY-1S, CDC-CYBER 205 and the Parallel Computer ICL-DAP. Architecture and Programming (in german). DFVLR-FB 82-02 Report, Koeln 1982.

[8] Gentzsch W.: Vectorization of Computer Programs with Aplications to Computational Fluid Dynamics. Vieweg Publ. Comp., Braunschweig [F.R.G.] 1984.

[9] Gentzsch W.: IBM 3090VF, Utilization and Performance Evaluation. Report Regensburg, Dec.1986.

[10] Gentzsch W.: An Introduction to the Vectorcomputer UNISYS ISP. Report Regensburg, June 1987.

[11] Higbie L.: Vectorization and Conversion of FORTRAN programs for the CRAY-1 [CFT] compiler. Pub. 2240207, CRAY Resarch, Minneapolis 1979.

[12] Hockney R.W.: Performance of parallel computers. Proc. NATO Advanced Research Workshop on High-Speed Computation, Juelich, 20-22 June, 1983.

[13] Kascic M.J.: Vector processing on the Cyber 200. Infotech State of the Art Report "Supercomputers", Infotech Int. Ltd., Maidenhead, U.K. 1979.

[14] Larson J.L.: Multitasking on the CRAY X-MP-2 Multiprocessor. Computer, 1984, 62-69.

[15] Rudsinki L., Worlton J.: The impact of scalar performance on vector and parallel processors. In: High Speed Computer and Algorithm Organization [Kuck,D.J., Lawrie, D.H., Sameh,A.H., eds.], Acad.Press, New York 1977, 451-452.

[16] Scarborough, R. G.: Kolsky, H.G.: A Vectorizing Fortran Compiler. IBM J. Res and Devel., Vol. 30, No 2, pp. 163 - 171, 1986.

[17] Schoenauer W.,Gentzsch W. (Eds.): The Efficient Use of Vectorcomputers with Emphasis on CFD. Vieweg Publ.,F.R.G. 1986.

CHAPTER 5: RESTRUCTURING OF BASIC LINEAR ALGEBRAIC ALGORITHMS (Prof. Dr. Gentzsch)

5.0 INTRODUCTION

In the previous chapter the vectorization of typical do-loops arising in CFD problems was discussed. Until now, we have not considered the algebraic structure of the basic algorithm. This is by far the most important consideration. Indeed, a good algorithm poorly coded is usually preferable to a poor one optimally coded. For this reason, in the following, we turn our attention to the restructuring of some basic linear algebraic algorithms such as matrix*vector and matrix*matrix operations, linear recursions up to more complex *algorithms and iterative methods for the solution of linear algebraic* systems of equations with sparse matrices. Problems involving linear algebraic calculations consume large quantities of computer time. If substantial improvements can be made within the arithmetic section, a significant reduction in the overall computation time will be realized.

5.1 BASIC VECTOR OPERATIONS

Many linear algebraic algorithms contain one or more of the following operations

vector * vector
matrix * vector
matrix * matrix.

The first one is easily vectorized by the current compilers but the latter two operations sometimes will cause problems. In their original form the operational structure is, more or less serial, depending on the architecture of the computer. The matrix*vector procedure

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots\cdots & A_{1N} \\ A_{21} & A_{22} & \cdots\cdots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots\cdots & A_{NN} \end{pmatrix} * \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_N \end{pmatrix}$$

results in

$$Y_i = \sum_{k=1}^{N} A_{ik} X_k \quad , \quad i = 1,2,\ldots,N .$$

Here A(I,K) are the elements of an N * N - matrix A, and X, Y are vectors with N components. In other words the i-th component of Y is calculated by

Y(I) = (I-th row of A) * X,

which for many vector computers (e.g. VP 200, CRAY-2) with N even is not suitable, since the elements are stored by columns and data are loaded with large stride N. But looking at the operation more globally

$$\begin{aligned} Y_1 &= A_{11}X_1 + A_{12}X_2 + \cdots + A_{1N}X_N \\ Y_2 &= A_{21}X_1 + A_{22}X_2 + \cdots + A_{2N}X_N \\ &\vdots \\ Y_N &= A_{N1}X_1 + A_{N2}X_2 + \cdots + A_{NN}X_N \end{aligned}$$

yields the following basic vector structure

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \\ \vdots \\ A_{N1} \end{pmatrix} *X_1 + \begin{pmatrix} A_{12} \\ A_{22} \\ \vdots \\ A_{N2} \end{pmatrix} *X_2 + \ldots + \begin{pmatrix} A_{1N} \\ A_{2N} \\ \vdots \\ A_{NN} \end{pmatrix} *X_N$$

The elements of each column of the matrix A are stored contiguously in the memory. The vectorized algorithm then has the form

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix} = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{pmatrix} + X_j * \begin{pmatrix} A_{1j} \\ A_{2j} \\ \vdots \\ A_{Nj} \end{pmatrix} \qquad \text{for } j = 1,2,\dots,N .$$

In a previous step all elements of the vector Y are set to zero. The result is a triadic (SAXPY) operation of the form

vector = vector + scalar * vector.

For many vector computers, an (assembler) library routine is available for this operation and the performance rate of these triads is almost doubled for long vectors.

For the matrix*vector case re-ordering of the corresponding FORTRAN-code is very simple. One only has to interchange the loop-indices I and J to transform a row-like algorithm into a column-like one to obtain linked triads:

Scalar loop:

```
   DO   1    I = 1,N
   DO   1    J = 1,N
 1 Y(I) = Y(I) + X(J) * A(I,J)
```

Vector loop:

```
   DO   1    J = 1,N
   DO   1    I = 1,N
 1 Y(I) = Y(I) + X(J) * A(I,J)
```

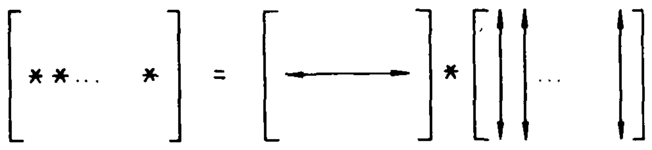
The first inner loop represents a product of two vectors X and A(I,*) while the vector inner loop results in a triad for each J.

This principle is easily carried over to more complex problems as we shall shortly demonstrate using the matrix multiplication for large, full matrices. Matrix multiplication has three loops: an inner, middle, and outer. In the inner loop the arithmetic operations take place. By fixing the inner loop expression and varying the loop indices I,J, and K, six variants are possible for arranging the three loop indices. Each differs in how access is to be made to the matrix elements, i.e. by row or by column, as a scalar, vector, or matrix. Each permutation will have a different memory access pattern, which will have an important impact on its performance on a vector processor. The following discussion of the variants is presented only for the purpose of demonstration. However, for real production codes, use of library routines for matrix multiplication is recommended.

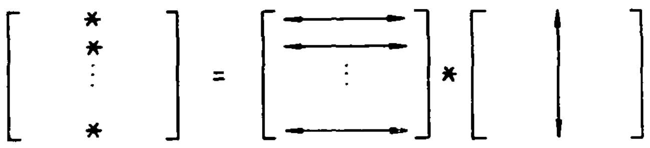
Given the N x N-matrices A and B, the formation of the product A * B = C in the usual manner is

```
   DO   1    I = 1,N
   DO   1    J = 1,N
   DO   1    K = 1,N
 1 C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

after having set C(I,J) = 0.0 for all I,J in a previous step. The vector operation and data organization is described graphically by means of a diagram introduced by Dongarra [23]

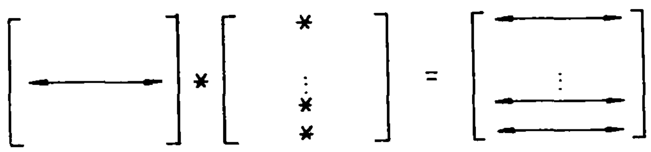


For the permutation of I and J, reference to the data is made slightly differently, so that the diagram is of the form

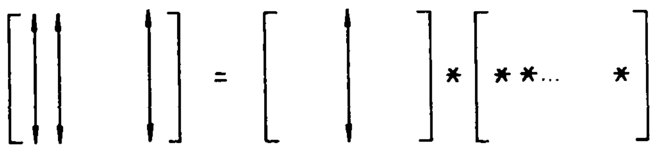


Both algorithms of the forms IJK and JIK are related by the fact that the inner loop is performing an inner product calculation. For reasons concerning bank conflicts when access is made to elements in a row of a matrix, the use of inner products is not recommended on machines similar to the CRAY-1. On the other hand, since an inner product machine instruction exists on the memory-to-memory machine CYBER 205, one might be tempted to use it. But this does not result in the quickest procedure.

The algorithms of the form KIJ and KJI are related in that they take a multiple of a vector and add it to another vector as a basic operation. For the form KIJ the access pattern appears as:

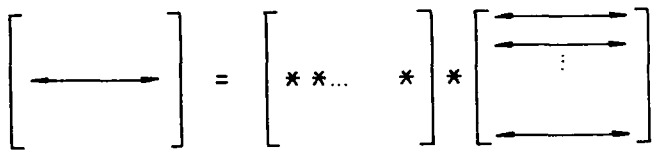


Every row of B is scaled by an element of A, and the result is used to update a row of C. For the form KJI we have:

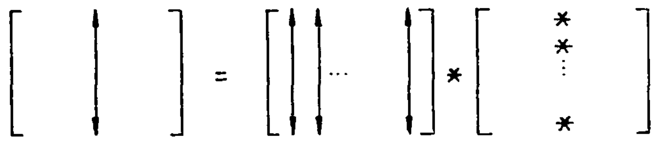


These algorithms are not to be recommended on vector computers since they up-date or load and store the entire matrix N-times for each pass through the outer loop.

The final two possibilities are IKJ and JKI. Here the access patterns appear as:



for IKJ and



for JKI. These forms are very efficient on the current vector computers for full matrices, when N is sufficiently large. They use multiples of a set of vectors and accumulate the results in a single vector before storing the vector. Both forms are suitable for a "CRAY-like" machine, while the latter is superior for "CYBER-like" machines due to the column-wise operational scheme. As an illustration, we will discuss the latter form JKI in more detail.

Instead of looking only at one element C(I,J) of the first form IJK, we write down all N results C(I,J) for fixed J:

$$\begin{aligned} C_{1J} &= A_{11}B_{1J} + A_{12}B_{2J} + \dots + A_{1N}B_{NJ} \\ C_{2J} &= A_{21}B_{1J} + A_{22}B_{2J} + \dots + A_{2N}B_{NJ} \\ &\vdots \\ C_{NJ} &= A_{N1}B_{1J} + A_{N2}B_{2J} + \dots + A_{NN}B_{NJ} \end{aligned}$$

Again a basic column-wise vector structure is recognizable:

$$\begin{pmatrix} C_{1J} \\ C_{2J} \\ \vdots \\ C_{NJ} \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \\ \vdots \\ A_{N1} \end{pmatrix} * B_{1J} + \dots + \begin{pmatrix} A_{1N} \\ A_{2N} \\ \vdots \\ A_{NN} \end{pmatrix} * B_{NJ}$$

for the *J-th column of C.* The vectorized algorithm then has the form

$$\begin{pmatrix} C_{1J} \\ C_{2J} \\ \vdots \\ C_{NJ} \end{pmatrix} = \begin{pmatrix} C_{1J} \\ C_{2J} \\ \vdots \\ C_{NJ} \end{pmatrix} + B_{KJ} * \begin{pmatrix} A_{1K} \\ A_{2K} \\ \vdots \\ A_{NK} \end{pmatrix} \qquad K = 1,2,\dots,N .$$

for every fixed J = 1,2,...,N,. For completeness, we add the following FORTRAN-version which corresponds to the above mentioned form JKI:

```
   DO   1   J = 1,N
   DO   1   K = 1,N
   DO   1   I = 1,N
 1 C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

B(K,J) is a constant for the innermost loop which again *represents a triadic operation*

vector = vector + vector * scalar.

5.2 MATRIX MULTIPLICATION FOR BANDED MATRICES

The main disadvantage of the usual approaches to matrix multiplication, discussed in the last section, is that they become inefficient for banded matrices with relatively narrow *bandwidths. We will demonstrate this fact* by means of the matrix multiplication of sparse tridiagonal matrices and the application of the form JKI which results in a pentadiagonal matrix with the column vectors

$$\underline{C}_K = A_{K-1,K} \cdot \underline{B}_{K-1} + A_{KK} \cdot \underline{B}_K + A_{K+1,K} \cdot \underline{B}_{K+1}$$

with

$$\underline{C}_K = \begin{pmatrix} C_{K-2,K} \\ C_{K-1,K} \\ C_{KK} \\ C_{K+1,K} \\ C_{K+2,K} \end{pmatrix}$$

and

$$\underline{B}_{K-1} = \begin{pmatrix} B_{K-2,K-1} \\ B_{K-1,K-1} \\ K_{K,K-1} \\ 0 \\ 0 \end{pmatrix} \qquad \underline{B}_{K} = \begin{pmatrix} 0 \\ B_{K-1,K} \\ B_{KK} \\ B_{K+1,K} \\ 0 \end{pmatrix} \qquad \underline{B}_{K+1} = \begin{pmatrix} 0 \\ 0 \\ B_{K,K+1} \\ B_{K+1,K+1} \\ B_{K+2,K+1} \end{pmatrix} .$$

Although the vector operations to evaluate the vector C can be treated as a linked triad for every K, namely

$$\underline{C}_K = \underline{C}_K + A_{IK}\,\underline{B}_I \quad , \quad I = K-1,K,K+1 \quad ,$$

the gain in performance is poor, resulting from the maximum vector length of 5 elements.

The key to the solution of this problem is to store the matrices by diagonals instead of by rows or columns. This storing seems natural for large banded matrices in the sense that the matrix is defined and stored in terms of very few vectors which are as long as possible. A second advantage is that the transpose of a matrix A is readily available in terms of the same diagonal vectors whose elements are stored consecutively in the memory.

The basic idea is that instead of forming a column of C, we will form a diagonal of C. Let us consider again the multiplication of tridiagonal matrices. The result is a penta-diagonal matrix

$$\underline{C}_{-2} \, , \; \underline{C}_{-1} \, , \; \underline{C}_{0} \, , \; \underline{C}_{1} \, , \; \underline{C}_{2}$$

where

$$\underline{C}_{-2} = \begin{pmatrix} 0 \\ 0 \\ C_{31} \\ \vdots \\ C_{K,K-2} \\ \vdots \\ C_{N,N-2} \end{pmatrix} , \quad \underline{C}_{-1} = \begin{pmatrix} 0 \\ C_{21} \\ \vdots \\ C_{K,K-1} \\ \vdots \\ C_{N,N-1} \end{pmatrix} , \quad \underline{C}_{0} = \begin{pmatrix} C_{11} \\ \vdots \\ C_{KK} \\ \vdots \\ C_{NN} \end{pmatrix}$$

$$\underline{C}_{1} = \begin{pmatrix} C_{12} \\ \vdots \\ C_{K,K+1} \\ \vdots \\ C_{N-1,N} \\ 0 \end{pmatrix} , \quad \underline{C}_{2} = \begin{pmatrix} C_{13} \\ \vdots \\ C_{K,K+2} \\ \vdots \\ C_{N-2,N} \\ 0 \\ 0 \end{pmatrix} .$$

The zeros in the vectors may be omitted but are used here to obtain a homogeneous structure. The diagonals of A are defined as follows

$$\underline{A}_{-1} = \begin{pmatrix} 0 \\ A_{21} \\ \vdots \\ A_{K,K-1} \\ \vdots \\ A_{N,N-1} \\ 0 \end{pmatrix} , \quad \underline{A}_0 = \begin{pmatrix} A_{11} \\ \vdots \\ A_{KK} \\ \vdots \\ A_{NN} \\ 0 \end{pmatrix} , \quad \underline{A}_1 = \begin{pmatrix} 0 \\ A_{12} \\ \vdots \\ A_{K,K+1} \\ \vdots \\ A_{N-1,N} \\ 0 \end{pmatrix}$$

and analogously for B. Using the nomenclature of [64], V(p;q) will denote the vector

$$V(p;q) = \begin{pmatrix} V_{p+1} \\ \vdots \\ V_{M-q} \end{pmatrix}$$

where M is the vector length of V. We say that V(p;q) has been obtained by displacing the vector V by p from the top and q from the bottom. The following result can be verified in a straightforward computation

$$\begin{aligned}
\underline{C}_{-2} &= \underline{A}_{-1}(2;1) \cdot \underline{B}_{-1}(1;2) \\
\underline{C}_{-1} &= \underline{A}_{-1}(1;1) \cdot \underline{B}_0(0;2) + \underline{A}_0(1;1) \cdot \underline{B}_{-1}(1;1) \\
\underline{C}_0 &= \underline{A}_{-1}(1;0) \cdot \underline{B}_1(1;0) + \underline{A}_0(0;1) \cdot \underline{B}_0(0;1) + \underline{A}_1(1;0) \cdot \underline{B}_{-1}(1;0) \\
\underline{C}_1 &= \underline{A}_0(0;2) \cdot \underline{B}_1(1;1) + \underline{A}_1(1;1) \cdot \underline{B}_0(1;1) \\
\underline{C}_2 &= \underline{A}_1(1;2) \cdot \underline{B}_1(2;1) \quad .
\end{aligned}$$

The algorithm is not restricted to the multiplication of tridiagonal matrices. Analogous algorithms for more general banded matrices may be found in [64]. There a "FORTRAN-like" algorithm can be found. For full matrices, the start-up costs are about three times higher for the diagonal algorithm compared to the conventional algorithms described above. But for narrow banded matrices the diagonal algorithm is much more efficient.

5.3 LINEAR AND NONLINEAR RECURRENCES

The vectorization of algorithms including linear and nonlinear recurrences such as Thomas-, Richtmyer- and Rusanov-algorithms (see also section 6.3 and 6.7) for the solution of large, linear systems of algebraic equations with tridiagonal or block tridiagonal matrices is in general not straightforward. However, to avoid interdependences between neighboring grid points in two- and three-dimensional problems, a so-called Red-Black or ZEBRA-pattern in one dimension (cf. section 5.4) and a parallel evaluation of the recurrences in the other dimension(s) solve the problem in a rather elementary way with only a few modifications to an already existing program. This then allows vector operations on long and contiguously stored vectors and arrays, and maintaining not only the efficiency of the corresponding recursive algorithm, but also the high potential of the vector machine.

In the following we shall solve systems of n algebraic equations Au = f with the tridiagonal matrix:

$$A = \begin{pmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & b_n & a_n \end{pmatrix}$$

which often arise in practice, e.g. when solving ordinary or partial differential equations with second order derivatives by discrete numerical methods. There are a number of related methods for solving this system serially in a time proportional to n. One of these methods is Gaussian elimination, which for tridiagonal systems reduces to the so-called Thomas algorithm. It is very efficient with 5n-4 arithmetic operations compared to n(n**2+3n-1)/3 operations for the complete Gaussian elimination. We explicitly assume, that the LU decomposition of A into the product of a lower triangular matrix L and an upper triangular matrix U exists. That is, A = L * U where:

$$L = \begin{pmatrix} 1 & & \\ \gamma_2 & 1 & \\ & \ddots & \ddots \\ & \gamma_n & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} \alpha_1 & c_1 & & \\ & \alpha_2 & c_2 & \\ & & \ddots & \ddots \\ & & \alpha_{n-1} & c_{n-1} \\ & & & \alpha_n \end{pmatrix}.$$

After computing L and U, it is relatively straightforward to solve the resulting triangular systems of equations:

$$Ly = f \ , \ Uu = y.$$

The whole algorithm can be expressed in three stages:

decomposition:

$$\alpha_1 = a_1$$

$$\gamma_i = b_i/\alpha_{i-1} \qquad i = 2,3,\dots,n$$

$$\alpha_i = a_i - c_{i-1}\gamma_i$$

forward substitution:

$$g_1 = f_1$$

$$g_i = f_i - \gamma_i g_{i-1} \qquad i = 2,3,\dots,n$$

backward substitution:

$$u_n = g_n/\alpha_n$$

$$u_i = (g_i - c_i u_{i+1})/\alpha_i \qquad i = n-1,\dots,1$$

The algorithm is stable, if

$$|a_1| > |c_1| > 0$$

$$|a_i| \geq |b_i| + |c_i| ,$$

$$b_i \cdot c_i \neq 0$$

$$|a_n| > |b_n| > 0 ,$$

which in many applications can be fulfilled. If one of the conditions:

$$c_1 \neq 0 , \quad b_i c_i \neq 0 , \quad b_n \neq 0$$

is violated, the system can be reduced to two or more smaller systems which are essentially uncoupled.

In solving systems of m partial differential equations there arise systems of algebraic equations of the above tridiagonal form, but with a block tridiagonal matrix A where the coefficients are now $m \times m$ -matrices. The numerical treatment of the boundary-layer equations for two-dimensional, incompressible, viscous flows leads to systems with 2x2-blocks, whereas 5x5-blocks arise in the solution of the Navier-Stokes equations for three-dimensional compressible viscous flows. In these cases the above mentioned Thomas algorithm extends easily to the Richtmyer algorithm. The vectorization of the Richtmyer algorithm will be explained in section 6.7 in further detail.

Unfortunately the three loops in the Thomas algorithm are all recurrences that must be evaluated one term at a time. The fact that the previous element must be known before the present one is computed, prevents the algorithm from taking any advantage of the vector hardware features on a computer since all elements of a vector must be known before a vector operation is initiated. Hence the algorithm which is the fastest one solving tridiagonal systems on a serial computer, is highly unsuitable on a vector computer.

However in all cases that occur in applications there are alternatives solving these problems [40]. A detailed description for a special example, the mesh generation algorithm of Thompson ([123], Chapter 6.8) is given in the next chapter.

5.4 *ITERATIVE ALGORITHMS*

In order to illustrate restructuring of iterative methods for the solution of algebraic equations resulting from the discretization of partial differential equations, we will consider the successive over-relaxation (SOR) method for the following model problem. Let f(x,y) and g(x,y) be continuous functions defined in the interior G and on the boundary dG, respectively, of the unit square

$$G = \{(x,y) \mid 0 < x < 1, \ 0 < y < 1\} .$$

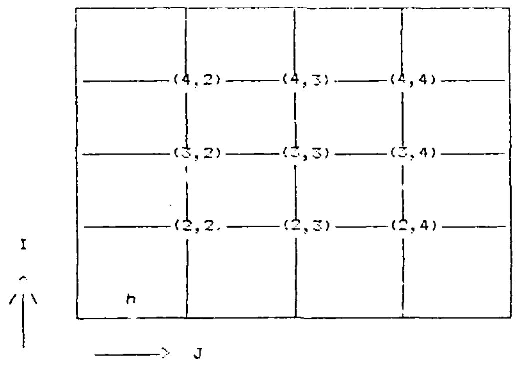
We are looking for a function u(x,y) continuous in G + dG, twice continuously differentiable in G, which satisfies Poisson's equation

$$\text{(1)} \quad \text{delta } u = \partial^2 u/ \partial x^2 + \partial^2 u/ \partial y^2 = f(x,y) \quad \text{in } G$$

and the boundary condition

$$u(x,y) = g(x,y) \quad \text{on } dG.$$

We shall primarily be concerned with the linear system of algebraic equations arising from the numerical solution of this model problem using two different five-point difference equations on a mesh of horizontal and vertical lines with mesh spacing $h = 1/(M-1)$ for some integer M:



For a given mesh point (x,y) we approximate the second derivatives of the model problem by the usual central difference quotients

$$\partial^2 u/\partial x^2 \approx [\ u(x+h,y) - 2\,u(x,y) + u(x-h,y)\ ]/\ h^2$$

(2)

$$\partial^2 u/\partial y^2 \approx [\ u(x,y+h) - 2\,u(x,y) + u(x,y-h)\ ]/\ h^2$$

and obtain, for each interior mesh point, one linear algebraic equation of the form

$$4*u(x,y) - u(x+h,y) - u(x-h,y) - u(x,y+h) - u(x,y-h) = -h^2 f(x,y)$$

With the ordering I = 1,2,..., M of the mesh points in x-, and J = 1,2,...,M in y-direction, this results in

$$(3)\quad 4*U(I,J) - U(I+1,J) - U(I-1,J) - U(I,J+1) - U(I,J-1) = -h^2 F(I,J)$$

for the interior mesh points we seek (M-2)*(M-2) approximate solutions U(I,J) of the (M-2)*(M-2) equations forming the linear system

$$(4)\quad Au = b$$

which, for small values of M, can be solved directly wit the aid of a special variant of Gaussian Elimination. Usually, however, such systems ar solved iteratively, and very often, the Successive Overrelaxation (SOR) method is used in engineering and scientific applications. Therefore, we choose this method as a basis for our discussion of different variants of SOR and their restructuring with respect to special vector computer architectures. As example, we consider (1) with $f(x,y) = 1$ and $g(x,y) = (x^2+y^2)/4$, with the exact solution $u(x,y) = (x^2+y^2)/4$. Let

$$A = D - L - U$$

be a special partitioning of the matrix A with diagonal matrix D, strictly lower triangular matrix L and strictly upper triangular matrix U. Thus, given the n-th iterate of u, one determines the (n+1)th iterate by

$$(5)\quad u^{(n+1)} = L_w u^{(n)} + (I - wL)^{-1} wD^{-1} b$$

with the identity matrix I, the iteration matrix

$$L_w = (I - \omega L)^{-1} \ (\omega U + (1-\omega) I)$$

and the relaxation factor ω for which $0 < \omega < 2$. For the model problem, the optimal relaxation factor is (see [99])

$$\Omega_{opt} = 1 + \left[\frac{\mu}{1 + (1 - \mu^2)^{1/2}} \right]^2$$

with $\mu = \cos \pi h$.

For programming purpose, it is more adequate to consider the algorithm point-wise :

$$U^{(n+1)}(I,J) = (1 - \omega) U^{(n)}(I,J) + 0.25\omega\, (U^{(n+1)}(I-1,J) + U^{(n+1)}(I,J-1) +$$

$$+ U^{(n)}(I+1,J) + U^{(n)}(I,J+1) - h^2F(I,J)) \tag{6}$$

for I,J = 2,3,..., M-1. For $\omega = 1$, SOR reduces to the Gauss - Seidel method

$$U^{(n+1)}(I,J) = 0.25 (U^{(n+1)}(I-1,J) + U^{(n+1)}(I,J-1) + U^{(n)}(I+1,J) +$$

$$U^{(n)}(I,J+1) - h^2F(I,J))$$

which results from equation (3) by using the most recently obtained approximate values for U(I-1,J), U(I,J-1). It is exactly this feature of the Gauss - Seidel and the SOR method which makes them not very suitable for vector computers: Vector dependences arise for both indices I and J. *The algorithm is recursive : U(I,J) depends on U(I-1,J) and* U(I,J-1). E.g. for fixed J,running with I from 2 to M-1, computing U(I,J) requires U(I-1,J) which is still in the pipelined functional unit. This dependence is recognized by the autovectorizing compiler and computation is forced to be performed on the scalar processor. Otherwise, on the vector processor, the algorithm would degenerate to the *very slowly converging Jacobi method.* Thus, the *original* SOR *version* (6) *is no longer* a powerful method on vector computers. Therefore, it is necessary to find out special structures of the method which fit the architecture of the vector computer and, furthermore, do not destroy the good convergence property of the method.

One possible modification is the diagonal - wise computation of the U(I,J). The corresponding path through the mesh is shown in the following figure for M = 5 :

```
   Loop 50

 ________________________
I     I     I     I     I
I_____I_____I_____I_____I
I   1 I   3 I   6 I     I
I_____I_____I_____I_____I
I   2 I   5 I   8 I     I
I_____I_____I_____I_____I
I   4 I   7 I   9 I     I      Loop 55
I_____I_____I_____I_____I
```

The mesh points 1-6 of the upper left triangle of the mesh including the main diagonal are computed in DO-loop 50, and the mesh points 7-9 of the lower right triangle are computed in DO-loop 55:

```
   DO      50      K = 3,5
   DO      50      J = 2,K-1
50 U(K-J+1,J) = ...

   DO      55      K = 2,3
   DO      55      J = 2,5-K
55 U(M-J+1,K+J-1) = ...
```

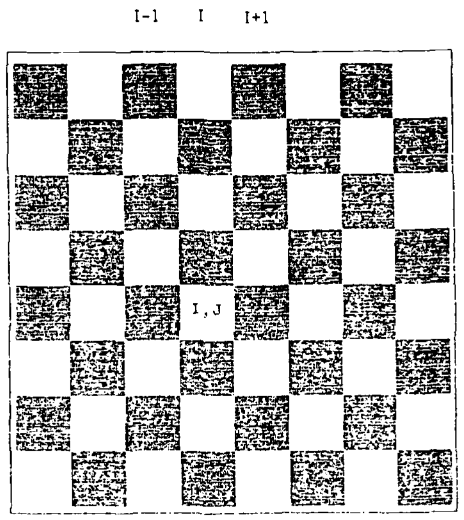
The inner loops are no longer recursive, which may be realized by a good vectorizing compiler. Indeed, U5 (which corresponds to U(3,3) in our example) depends on U2, U3, U7 and U8 , but neither on U4 nor on U6 of the same diagonal.

At a first glance, this restructuring seems to be the solution of our vectorization problem. However, this variant involves very short vectors (worst case: vector length of 1!),and data needed in the inner loops has to be gathered from memory with large stride M-1 which, for some vector computers, decreases performance dramatically. Therefore, this variant cannot be recommended for vector computers.

Next we discuss the efficient restructuring of the classical SOR method (6) with respect to special vector architectures. As a first basis for vectorization we use red-black ordering of the method (also called checkerboard SOR).

DIFFERENT VARIANTS OF RED - BLACK SOR

Consider a checkerboard with red and black areas each representing one point of our computational mesh. Each red point has four black neighbors and each black point has four red neighbors.



Thus, computation of a "red" U(I,J) in formula (6) involves only the four black values

U(I-1,J), U(I+1,J), U(I,J-1) and U(I,J+1),

and vice versa. Red-black SOR, therefore, in a first half step computes all the red values using only old black values, and in a second half step computes all the black values using the red values of the first half step. In the literature (see [99]) this method is a special case of the so-called modified SOR method. The matrix A of (4) has the special form

$$(7) \quad A = \begin{bmatrix} D1 & H1 \\ H2 & D2 \end{bmatrix}$$

where D1 and D2 are square nonsingular diagonal matrices, D1 belonging to the red equations and D2 to the black equations. If we partition u and b in (4) in accordance with the partitioning of A, we can write the system (4) in the form

$$\begin{bmatrix} D1 & H1 \\ H2 & D2 \end{bmatrix} * \begin{bmatrix} u1 \\ u2 \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \end{bmatrix}$$

or

$$\begin{aligned} &D1*u1 + H1*u2 = b1 \\ &H2*u1 + D2*u2 = b2 \end{aligned} \qquad (8)$$

Evidently (8) is equivalent to the system

$$\begin{aligned} &u1 = F1*u2 + c1 \\ &u2 = F2*u1 + c2 \end{aligned} \qquad (9)$$

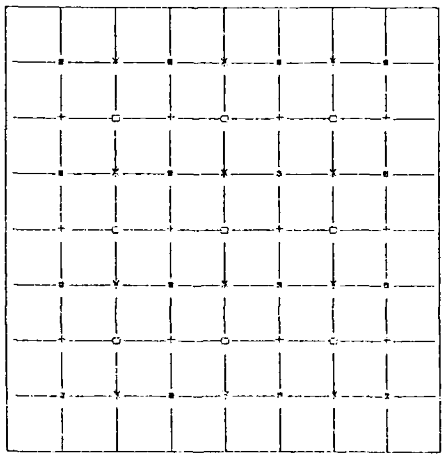
where

$$F1 = -D1^{-1}*H1,\ F2 = -D2^{-1}*H2,\ c1 = D1^{-1}*b1,\ c2 = D2^{-1}*b2\ .$$

System (9) consists of two half steps mentioned earlier, with vector u1 containing all red values and u2 containing all black values. For programming purpose, the following partitioning is more suitable and easily implemented:

```
   Do   2   J = 2, M-1, 2
   DO   2   I = 2, M-1, 2
2  U(I,J) = ...

   DO   3   J = 3, M-2, 2
   DO   3   I = 3, M-2, 2
3  U(I,J) = ...

   DO   4   J = 2, M-1, 2
   DO   4   I = 3, M-2, 2
4  U(I,J) = ...

   DO   5   J = 3, M-2, 2
   DO   5   I = 2, M-1, 2
5  U(I,J) = ...
```

where M is an odd integer (in applications, M-1 often is a power of 2). The arrangement of the gridpoints is shown in the following figure:



▪	Loop 2
○	Loop 3
+	Loop 4
*	Loop 5

However, red-black SOR has several disadvantages, with respect to vector computers, above all

* vector length is only (M-1)/2
* data are gathered with stride 2.

For most vector computers, short vector length and / or stride 2 reduces performance drastically. Therefore, it is recommended to avoid short vectors and stride 2, if possible.

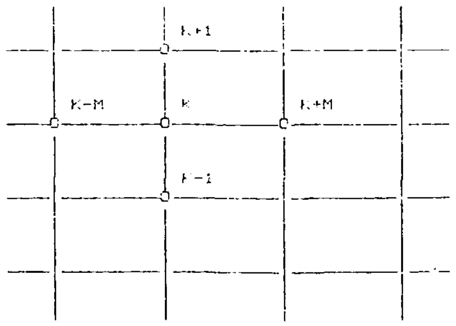
One solution to avoid short vectors is to perform computations under the control of a logical mask. The construction of the masks for red-black SOR would look like

```
   DO   1   J = 1,M
   DO   1   I = 1,M
   MASKRED (I,J) =  .TRUE.
   MASKBLA (I,J) =  .TRUE.
1  CONTINUE

   DO   2   J = 2, M-1,2
   DO   2   I = 2, M-1,2
2  MASKRED (I,J) =  .FALSE.

   DO   3   J = 3, M-2,2
   DO   3   I = 3, M-2,2
3  MASKRED (I,J) =  .FALSE.

   DO   4   J = 2, M-1.2
   DO   4   I = 3, M-2,2
4  MASKBLA (I,J) =  .FALSE.

   DO   5   J = 3, M-2,2
   DO   5   I = 2, M-1,2
5  MASKBLA (I,J) =  .FALSE.
```

This has to be done once at the very beginning of the main program. In the subroutine then, the two-dimensional arrays have to be equivalenced with long one-dimensional arrays. The main part of the algorithm, for the red values, has the following form

```
   DO   2   K = M22, MM1
   IF (MASKR (K) )  GOTO 2
   U(K) = (1-Ω) * U(K) + 0.25*Ω*
   (U(K-M) + U(K-1) + U(K+1) + U(K+M) + HH)
2  CONTINUE
```

with M22 = M+2, MM1 = (M-1) * M and HH = $-h^2$, and analogously for the black values.



If there is a vector mask register in the vector processor, where the 0- and 1-bits of the logical masks can be stored, the computations are performed under the control of these masks for vectors with a length of about (M-1)*(M-1)/2.

However, these masked operations are time-consuming. To avoid this, the red and the black values have to be stored contiguously into red and black vectors, respectively, by hand. We will demonstrate this for a one - and a two - dimensional version.

Let us begin with the two - dimensional version. Four contiguous vectors U1, U2, U3 and U4 are constructed for the elements U(I,J) of the DO- loops 2,3,4 and 5, respectively. As example, we show the principle for the first and second loop computing red values from black ones:

```
   L = 1
   DO   21   J = 2, M-1,2
   K = 1
   L = L + 1
       DO   21   I = 2, M-1,2
       K  = K+1
       U1 (K,L) = U (I,J)
21 CONTINUE
```

The next loop involves also the red boundary points:

```
   L = 0
   DO   22   J = 1,M,2
   K = 0
   L = L+1
       DO   22   I = 1,M,2
       K  = K+1
       U2 (K,L) = U (I,J)
22 CONTINUE
```

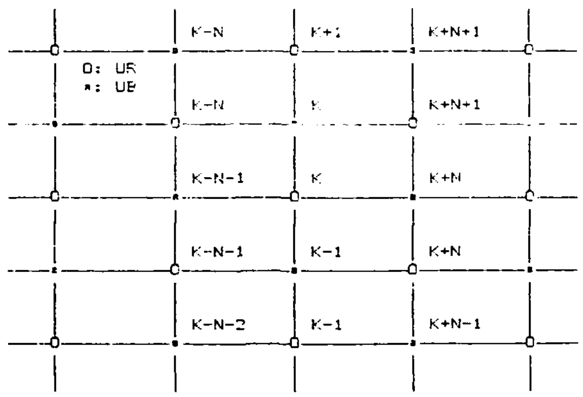
A similar structure also holds for the last two loops computing the black values. After this preparational phase, the iteration procedure starts. For the computation of U1 (I,J), e.g., we get

```
   DO   50   J = 2,N
   DO   50   I = 2,N
   U1 (I,J) = (1-Ω) * U1 (I,J) + 0.25 *Ω* (U4 (I.J-1) +
   U4 (I,J) + U3 (I-1,J) + U3 (I,J) + HH)
50 CONTINUE
```

where N = (M-1)/2+1 is the vector length of U1 along a line y = const. After the required accuracy has been reached, the elements of U1, U2, U3 and U4 have to be restored into U and the algorithm is complete.

Similarly cumbersome is the fully one - dimensional red-black version with stride 1. However, when storing the red values in *one long red vector UR with elements UR(K) and* running with K from KSTART to KLAST, the boundary values are overwritten by the corresponding computations. Therefore, after each K-loop, the original boundary values have to be restored into UR. The same holds for the black vector UB. Here, the gathering of the black values looks like

```
   K = 0
   DO   30   J = 1,M,2
   DO   31   I = 1,M,2
   K = K + 1
   UB (K) = U (I,J)
   UBR(K) = U (I,J)
31 CONTINUE
   IF (J.EQ.M)   GOTO   30
   DO   32   I = 2,M-1,2
   K = K + 1
   UB (K) = U (I,J+1)
   UBR(K) = U (I,J+1)
32 CONTINUE
30 CONTINUE
```



UBR is used for checking the accuracy and for saving the boundary values. A similar preparation holds for UR and URR. Next the iteration procedure follows with the update of the black and the red values, e.g.

```
   DO   50   K = N2, NM
   UB (K) = (1-w) * UB (K) + 0.25 *w* (UR (K-N-1) + UR (K+N)
          + UR (K-1) + UR (K) + HH)
50 CONTINUE
```

where N2 = N+2, N = (M-1)/2, NM = N * M.

Again, the black and the red values UB and UR, respectively, have to be restored into U after the iteration procedure has been completed.

Clearly, this one - dimensional long - vector version is the *fastest red-black SOR* variant for all vector computers. It is very suitable for those vector computers with long start - up times and / or stride problems,e.g. the CYBER 205 and the CRAY-2. Restructuring two - dimensional arrays into long one - dimensional vectors, however, is often very cumbersome, above all for more complicated problems in engineering applications. Thus where possible, this kind of restructuring has to be avoided. In the case of SOR a new variant is recommended, [43] , which

* avoids stride problems
* preserves long vectors
* is easily implemented
* converges faster than SOR and red-black SOR.

A FULLY VECTORIZABLE SOR VARIANT

The model problem (1) can be discretized not only by the five-point difference equation (3) but also by the following version:

(10) 4*U(I,J) - U(I+1,J+1) - U(I-1,J+1) - U(I+1,J-1) - U(I-1,J-1) = - $2h^2$*F(I,J)

which may be obtained when rotating the difference operator (3) or using the transformation

(11) $\bar{x}$ = (y+x)/SQRT(2) , $\bar{y}$ = (y-x)/SQRT(2)

Because of the structure of the difference star (" X "), we will call (10) in combination with SOR the XSOR method.

The accuracy of the approximations (3) and (10) is similar: A Taylor series expansion yields (with n = 4) the truncation errors of the corresponding difference equations are

$$- h^2 * (\partial nu/\partial xn + \partial nu/\partial yn) / 12$$

and

$$- h^2 * (\partial nu/\partial xn + 6\, \partial nu/\partial x^2 dy^2 + \partial nu/\partial yn)/ 12$$

for (3) and (10), respectively, Therefore, the order of accuracy for both approximations is $O(h^2)$ with the same coefficient $-h^2/12$ of the fourth order derivatives plus, in addition a mixed derivative term which may cause a reduction in grid size h to obtain comparable accuracy.

The matrix A of the resulting system of equations is a symmetric M-matrix [99], and therefore, XSOR converges for all relaxation parameters w with $0< w <2$. Let again

$$A = D - L - U$$

be a special partitioning of A with diagonal matrix D, strictly lower triangular matrix L and strictly upper triangular matrix . Then, the maximum eigenvalue of the corresponding matrix

$$(12) \quad B = D^{-1} *(L+U)$$

leads to the asymptotic rate of convergence of SOR as is defined in [99] . The eigenvalues f of B are computed from

$$\det(\mu I - B) = 0$$

or, equivalently,

$$(13)\ \mu V(x,y) = [V(x+h,y+h)+V(x+h,y-h)+V(x-h,y+h)+V(x-h,y-h)]/4$$

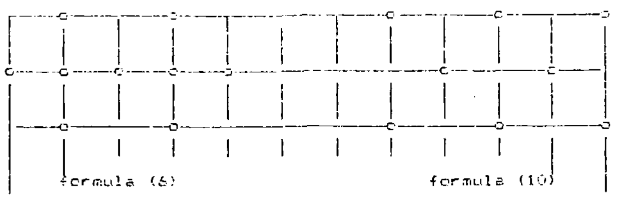
which yields

$$(14) \quad \mu_{p,q} = \cos p\pi h * \sin q\pi h$$

for $p,q = 1,2,\dots, M-1$. Thus, the maximum eigenvalue of B is

$$\mu_{max} = \cos \pi h \approx 1 - 0.5\pi^2h^2$$

which is the same as for the classical SOR method, so that the asymptotic rate of convergence for both methods is identical. In practice, however, convergence of the XSOR variant is faster Than that of (6): Near the boundary, (6) involves one boundary Point and two at the corners, while formula (10) involves two boundary points and three at the corners:



Therefore, the XSOR variant with the discretization (10) is faster than (6) and even faster than red-black SOR and ZEBRA-line SOR as described in [39],[40].

The implementation of the new XSOR variant is easy. Only minor changes of the indices of (6) are necessary.

Now, for fixed index J, running with I from 2 to M-1, U(I,J) no longer depends on the previously computed U(I-1,J). Therefore, all values U(I,J) for fixed J can be computed contiguously on the vector processor, as they are stored in main memory.

Some vector compilers do not recognize that the U(I,J) for fixed J are independent, and the computation is performed on the scalar processor. Here, compiler directives forcing vectorization are applied successfully.

Only for very short vectors (depending on the vector computer) a further remarkable improvement in performance may be achieved with the aid of some modifications. In the following, we will briefly discuss several variants such as

* ZEBRA - XSOR, without and with loop-unrolling,
* ZEBRA - XSOR, under control of a logical mask,
* ZEBRA - XSOR, hand-coded one-dimensional version.

For ZEBRA - XSOR the computational mesh is divided into two sets of mesh points, namely red and black points. For J even, all the points are defined to be red, and for J odd, all the points are defined to be black. Then, the algorithm (in principle similar to red-black SOR) looks like

```
   DO   50   J = 2, M-1,2
   DO   50   I = 2, M-1
50 U (I,J) = ...

   DO   51   J = 3, M-2,2
   DO   51   I = 2, M-1
51 U (I,J) = ...
```

For several vector computers, having vector registers as fast storage devices, it is sometimes useful to keep data as long as possible in these registers avoiding time-consuming load and store operations. Within one ZEBRA cycle,e.g., the elements U(I,4), I = 2,3,..., M-1, are needed for the computation of U(I,3) and U(I,5). Gathering the computation of the U(I,3) and the U(I,5) into one loop, avoids one load of the vector U(I,4). In the literature this is called "Loop-unrolling" with a depth of 2. Computing U(I,3), U(I,5), U(I,7) and U(I,9) for I = 2,3,..., M-1 in one inner loop leads to loop-unrolling with a depth of four and saves three load (for U(I,4), U(I,6) and U(I,8)). The structure of the ZEBRA - XSOR algorithm unrolled with a depth of 2 looks like

```
   DO   50   J = 2, M-1,4
   DO   50   I = 2, M-1
   U (I,J)   = ...
   U (I,J+2) = ...
50 CONTINUE

   DO   60   J = 3, M-4,4
   DO   60   I = 2, M-1
   U (I,J)   = ...
   U (I,J+2) = ...
60 CONTINUE

   DO   61   I = 2, M-1
61 U (I,M-2) = ...
```

Loop 50 updates all the red values while loop 60 and loop 61 update all the black values. Loop 61 is necessary in the case where M-1 is a power of 2. Other modifications are necessary if M-1 is not a power of 2.

Two-dimensional problems involving very short vectors (vector length, e.g., less than about 20) should be restructured into one-dimensional long-vector versions as is shown for the classical red-black SOR algorithm in the previous chapter. Again, for ZEBRA - XSOR, computations are performed under the control of logical masks. The masks are prepared in the main program. There, the elements

```
   MASKB (I,J)
```

are .TRUE. for J even and on the boundary (and .FALSE. otherwise) and the elements

```
   MASKR (I,J)
```

are .TRUE. for J odd and on the boundary (and .FALSE. otherwise). In the corresponding subroutine, the two-dimensional array U has to be equivalenced with the long one-dimensional array UD. Then, the main part of the algorithm has about the following form (with M22 = M+2, MM1 = (M-1) * M, MM = 2*M+1 and MM2 = (M-2) * M)

```
   DO   50   K = M22, MM1
   IF (MASKR(K))   GOTO 50
   UD (K) = ...
50 CONTINUE

   DO   55   K = MM, MM2
   IF (MASKB(K))   GOTO 55
   UD (K) = ...
55 CONTINUE
```

In cases where logical masks are expensive (which may easily be tested with the aid of two or three simple kernels), hand coding the two-dimensional problem into a one-dimensional structure is necessary. This, again, is only recommended for problems with very short vectors (vector length less than about 20 - 40, depending on the vector computer). Values U(I,J) for J even (red) have to be stored explicitly into a one-dimensional long vector UR, and all values U(I,J) for J odd (black) have to be stored explicitly into a one-dimensional long vector UB :

```
   K = 0
   DO   30   J = 1, M,2
   DO   30   I = 1, M
   K = K + 1
   UB (K) = U (I,J)
   UBR (K)= U (I,J)
30 CONTINUE

   K = 0
   DO   32   J = 1, M-2,2
   DO   32   I = 1, M
   K = K + 1
   UR (K) = U (I,J+1)
   URR (K)= U (I,J+1)
32 CONTINUE
```

Again, as in the hand-coded long-vector version of red-black SOR, UBR and URR are used for checking the accuracy and for saving the boundary values which, in every iteration step, are overwritten during computation of UR and UB:

```
   DO   40   K = M22, MM2
40 UBR (K) = UB (K)
   DO   50   K = M22, MM2
50 UB (K)  = ...
   DO   51   K = MM, MM4, M
   UB (K)  = UBR (K)
51 UB (K+1)= UBR (K+1)
   DO   52   K = 2, MM2
52 URR (K) = UR (K)

   DO   55   K = 2, MM2
55 UR (K)  = ...
   DO   56   K = M, MM4, M
   UR (K)  = URR (K)
56 UR (K+1)= URR (K+1)
```

with M22 = M+2, MM = 2 * M, MM2 = (M-1) * M/2-1 and MM4 =(M-1) * M/2 - M, where M-1 is a power of 2. In loops 40 and 52, values of the previous iteration level are stored into UBR and URR, respectively. In loops 50 and 55, black values are computed from red values and vice versa. Finally in loops 51 and 56, the "false" boundary values in UB and UR (at the horizontal lines y = 0 and y = 1) are overwritten by the correct ones. Again, the implementation of this one-dimensional long-vector version for the new XSOR variant is by far less cumbersome than for the classical red-black SOR.

RESULTS

For completion, we present some results for the vector computers IBM 3090 - 200 with vector features (VF) and for the CRAY-2, together with a brief discussion. On these machines, the following methods have been implemented:

SOROLD:	Classical SOR method, highly recursive, structure (3)
SORB1 :	Red-black SOR, checkerboard pattern
SORB2 :	SORB1 with gathering of red values in red vectors and black values in black vectors which yields stride one computations
SORB3 :	SORB2, collecting all red vectors in one long red vector and all black vectors in one long black vector resulting in a one-dimensional long-vector version
SORV :	Fully vectorizable SOR method, structure (10)
SORV1 :	SORV combined with SORB1
SORV2 :	SORV combined with SORB2
SORV3 :	SORV combined with SORB3

All methods solve the model problem (1) for the Poisson equation in a quadratic domain. Table 1 gives timings (in seconds) for the IBM 3090VF, table 2 for the CRAY-2, and table 3 presents MFLOPS for the two machines (for only one processor each).

Table 4.8: Timings (in seconds) for the IBM 3090-200VF

	N = 31	N = 63	N = 127
SOROLD	0.089	1.46	23.27
SORV	0.041	0.31	5.20
SORB1	0.064	0.88	16.33
SORV1	0.064	0.42	8.21
SORB2	0.065	1.02	13.54
SORV2	0.065	0.47	6.49
SORB3	0.023	0.45	8.56
SORV3	0.025	0.26	5.30

Table 4.9: Timings (in seconds) for the CRAY-2

	N = 31	N = 63	N = 127
SOROLD	0.098	1.61	27.55
SORV	0.012	0.08	1.23
SORB1	0.024	0.35	4.60
SORV1	0.024	0.17	2.25
SORB2	0.017	0.21	2.19
SORV2	0.017	0.10	1.15
SORB3	0.007	0.14	1.98
SORV3	0.008	0.07	1.14

Table 4.10: MFLOPS for the IBM 3090-200VF and the CRAY-2

	N = 31		N = 63		N = 127	
	IBM	CRAY	IBM	CRAY	IBM	CRAY
SOROLD	11.1	10.1	11.4	10.4	10.9	9.2
SORV	20.3	70.1	25.9	95.4	25.3	107.0
SORB1	11.8	31.8	18.4	46.3	15.2	53.9
SORV1	11.9	32.4	18.3	45.6	15.7	57.3
SORB2	11.5	45.9	15.9	76.9	18.3	113.3
SORV2	11.8	44.7	16.4	74.6	19.9	112.0
SORB3	31.8	108.9	35.9	118.8	29.0	125.3
SORV3	29.4	97.2	29.4	109.7	24.3	113.1

* SORV is up to four times faster than SOROLD and up to three times faster than SORB on the IBM.

* SORV is up to 22 times faster than SOROLD and up to four times faster than SORB on the CRAY-2.

* SORV is much easier to implement than any other of the SOR variants.

* For the IBM machine, programming a two-dimensional problem in a one-dimensional structure is only recommended for very short vector length (less than about 30).
* For the CRAY machine, one-dimensional restructuring for all vector lengths yields much higher performance.
* For the CRAY-2, stride two is a much bigger problem than for the IBM.
* For problems of low vectorizability (e.g. SOROLD), both machines perform similarly.
* For highly vectorizable problems, the CRAY-2 is up to six times faster than the IBM 3090VF.

5.5 GROUP ITERATIVE METHODS

For group or block iterative methods, groups or blocks of unknowns are improved simultaneously. The blocks of unknowns to be improved simultaneously are determined by a partitioning imposed on the coefficient matrix. Examples are Line Jacobi (LJAC), Successive Line Over-Relaxation (LSOR) and Alternating Direction Implicit (ADI) methods which are widely used in engineering codes. We discuss the implementation of this class of algorithms using the LSOR method. The systems

$$A_{jj}\,U_j^{(\nu+1)} = P_j \;,$$

result from the simultaneous solution of the equations along grid lines Y = const. of the discrete region. For the model problem of section 5.4

$$P_j = (1-\omega)A_{jj}\,U_j^{(\nu)} + \omega\{U_{j-1}^{(\nu+1)} + U_{j+1}^{(\nu)} - HH\} \;,$$

where

$$A_{jj} = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & \ddots & \ddots & \ddots \\ & & -1 & 4 \end{pmatrix} \;,$$

HH means H*H*I with the identity matrix I and Uj and Pj are vectors at iteration level μ resp. μ+1. The optimum value of the relaxation factor for this problem is:

$$\omega_0 = \frac{2(1+2a^2)}{(1+\sqrt{2}a)^2}$$

with a = sinπh/2. The systems may be solved by the Thomas algorithm (LU decomposition for tri-diagonal matrices) presented in section 5.3. The first problem is that the vector Uj(μ+1) is a function of U(j-1)(μ+1) which may be still in the pipe when it is needed; so, to avoid interdependences of neighboring grid lines, a ZEBRA pattern is introduced resulting in an even-odd structure of the lines y = const. within each iteration step:

```
   J1 = 2 , J2 = N1
   DO   1   K  = 1,2
   DO   2   J  = J1,J2,2
   DO   2   I  = 2,N1
   P(I,J) = ... arithmetic ...
 2 CONTINUE
   J1 = 3 , J2 = N2
 1 CONTINUE
```

with N1 = N-1, N2 = N-2. For K=1 (K=2), the Thomas algorithms are carried out for all lines with even (odd) indices. Unfortunately the Thomas-algorithms contain three highly serial recursions, as pointed out in chapter 5.3. But if we simultaneously solve all the "black" systems, and afterwards all the "white" systems, the effectiveness of the algorithm is saved. All that is left to do is to interchange I and J. For the example, this results in:

```
   I1 = 2 , I2 = N1
   DO   1   K  = 1,2
   DO   2   J  = 2,N1
   DO   2   I  = I1,I2,2
 2 P(I,J) = ... arithmetic ...
   I1 = 3 , I2 = N2
 1 CONTINUE
```

Without ZEBRA and for omega = 1, this algorithm degrades to Line Jacobi. The interchange of I and J is necessary to run over the first index of P as it is stored in memory.

An application for this, a mesh-generation problem, is discussed in detail in the next chapter.

5.6 CYCLIC REDUCTION

The principle of cyclic reduction may also be applied to the direct solution of the general tridiagonal system of equations. This direct method, which is very suitable for vector computers (cf. [47], [59], [75]), was originally developed by Hockney [49] and has been used in direct solutions to Poisson's equation.

The basic idea is to eliminate the odd-subscripted unknowns from the even-numbered equations and then re-group the even-numbered equations together by row and column permutations to generate another tridiagonal system of lower order. We will assume for simplicity that the order n of the system is a power of 2, although the algorithm works for arbitrary n. By way of an example, we shall consider a system Au = f for n = 8 and the 8x8-matrix:

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}.$$

The first step of the algorithm is to add a multiple (here 0.5) of an odd row to the even row below it

```
(odd)   -1     2    -1
(even)        -1     2     -1
```

so achieving the following:

```
(odd)   -1     2    -1
(even)  -0.5   0     1.5   -1
```

In the second step we add a multiple (here 0.5) of an odd row to the even row above it:

```
(even)  -0.5   0     1.5    -1
(odd)               -1       2    -1
```

so obtaining the following:

```
(even)  -0.5   0     1       0    -0.5
(odd)               -1       2    -1
```

The even-numbered rows now form another tridiagonal system of equations involving only the even-subscripted variables:

$$\begin{pmatrix} 1 & -0.5 & & \\ -0.5 & 1 & -0.5 & \\ & -0.5 & 1 & -0.5 \\ & & -0.5 & 1.5 \end{pmatrix} \begin{pmatrix} U_2 \\ U_4 \\ U_6 \\ U_8 \end{pmatrix} = \begin{pmatrix} f_2 + 0.5 \cdot (f_1+f_3) \\ f_4 + 0.5 \cdot (f_3+f_5) \\ f_6 + 0.5 \cdot (f_5+f_7) \\ f_8 + 0.5 \cdot f_7 \end{pmatrix}$$

The original size of the system was n = 2*2*2 . The smaller system now has size 2*2 . The odd-subscripted variables can be computed directly by substituting the known values of U2, U4, U6 and U8 into the odd-numbered equations, e.g.

$$U_3 = 0.5 * (f_3 + U_2 + U_4).$$

However, the explicit solution of the 2*2-system is not necessary since the above procedure can be repeated on this smaller system:

(odd)	-0.5	1	-0.5		
(even)		-0.5	1	-0.5	

resulting in:

(odd)	-0.5	1	-0.5		
(even)	-0.25	0	0.75	-0.5	

and similarly:

(even)	-0.25	0	0.75	-0.5	
(odd)			-0.5	1	-0.5

resulting in:

(even)	-0.25	0	0.5	0	-0.25
(odd)			-0.5	1	-0.5

Here the smaller system of half the size has a somewhat different form owing to its smaller size:

$$\begin{pmatrix} 0.5 & -0.25 \\ -0.25 & 1.25 \end{pmatrix} \begin{pmatrix} U_4 \\ U_8 \end{pmatrix} = \begin{pmatrix} r_4 \\ r_8 \end{pmatrix}$$

with

$$r_4 = f_4 + 0.75*(f_3+f_5) + 0.5*(f_2+f_6) + 0.25*(f_1+f_7)$$

$$r_8 = f_8 + 0.75*f_7 + 0.5*f_6 + 0.25*f_5$$

This process continues until one gets a single equation in one unknown. Again we have to add a multiple (here 0.5) of the first row to the second row

(odd)	0.5	-0.25
(even)	-0.25	1.25

so that, after three steps, one equation in one unknown, namely U8 ,remains:

$$1.125 * U_8 = r_8 + 0.5 * r_4 \; .$$

Having determined U8, substituting back into the other equations can proceed. U8 is substituted into the first equation of the 2x2-system and U4 and U8 are substituted into the first and third equation of the 4x4-system , so computing all the odd-subscripted variables from the original system and the algorithm is terminated.

A general derivation of cyclic reduction for systems Au = f with a tridiagonal matrix A is now straightforward. We carry out the same procedure described above to eliminate references to odd-subscripted variables in the even-numbered equations, using the following elementary row operations:

$$-\alpha_{2i}*\mathrm{Row}(2i-1) + \mathrm{Row}(2i) - \beta_{2i}*\mathrm{Row}(2i+1)$$

with $\alpha 2i = b2i/a2i-1$ and $\beta 2i = c2i/a2i+1$. Then the modified tridiagonal system of equations becomes:

$$-(b_{2i-1}\alpha_{2i})*U_{2i-2}+(a_{2i}-c_{2i-1}\alpha_{2i}-b_{2i+1}\beta_{2i})U_{2i}-c_{2i+1}\beta_{2i}*U_{2i+2}=$$

$$=f_{2i}-\alpha_{2i}f_{2i-1}-\beta_{2i}f_{2i+1} \qquad \text{for } i=1,2,\dots,2^{K-1}$$

where b1 = cn = 0. In general, each step of cyclic reduction reduces a (2**k)*(2**k)-system to one of size 2**(k-1)*2**(k-1), and after k steps we obtain one equation for the unknown U2 . Now substitution similar to that described above, yields the final result.

A vectorization of the reduction steps is now straightforward if the original matrix A is stored by diagonals. An implementation of the algorithm on vector computers is discussed in detail in [59].

5.7 SYSTEMS OF NON-LINEAR EQUATIONS

We now proceed to a short description of iterative methods for the solution of systems of n non-linear equations of the form:

```
f1(x1,x2,...,xn) = 0
f2(x1,x2,...,xn) = 0

     ...

fn(x1,x2,...,xn) = 0
```

which is also written in vector notation as:

$$f(x) = 0$$

with f= (f1,f2,...,fn) and x = (x1,x2,...,xn) . In general, the solution of non-linear systems for large n presents some severe difficulties concerning the correct initial data and convergence towards an unique solution x (if it exists). Let the above system be transformed into the equivalent form:

$$x = g(x) .$$

One well-known, nonlinear iteration method then is the generalized Newton method:

$$x^{(k+1)} = g(x^{(k)}) = x^{(k)} - J^{-1}(x^{(k)}) * f(x^{(k)})$$

where J(x) is the Jacobian matrix defined by

$$(J(x))_{ij} = \partial f_i(x)/\partial x_j .$$

It can be shown by means of a Taylor s series expansion about the exact solution x that under the circumstances that if f(x) has continuous second partial derivatives and the Jacobian matrix is nonsingular at x and if the iteration process determined by the generalized Newton method converges at all, it converges quadratically.

Instead of evaluating J(x(k)) and computing the inverse at each iteration step, we solve the equivalent sequence of linear systems

$$J(x^{(k)})\ x^{(k+1)} = J(x^{(k)})\ x^{(k)} - f(x^{(k)})$$

provided the Jacobian is non-singular. The matrix J(x(k)) and the right-hand side are known. These systems must be solved by direct or iterative linear system solvers. Vectorization then takes place according to the procedures described in sections 5.3-5.5 and chapter 6 and therefore will not be discussed further here.

In practice the Jacobian J(x) of f(x), which should be evaluated at x(k), will not be re-evaluated at each iteration, but only when necessary to ensure convergence. That is, if the iteration is converging with fixed J, the time-consuming process of re-evaluating and triangulating J is omitted.

To illustrate the vectorization of this class of methods, let

$$x^{(k)i} = (x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}, x_i^{(k)}, \dots, x_n^{(k)}) .$$

An iterative scheme of the form:

$$x^{(k+1)} = x^{(k)} - P^k(x^{(k)i}) \cdot f(x^{(k)i}) \quad , \quad k = 0,1,\dots$$

for i = 1,2,...,n is called to be of Gauss-Seidel type. The n*n matrix Pk depends on the algorithm used. In the important case of diagonal matrices Pk, the iterative scheme can be written componentwise as:

$$x_i^{(k+1)} = x_i^{(k)} - P_i(x^{(k)i}) f_i(x^{(k)i}) \quad , \quad k = 0,1,\dots$$

As an example, consider the stationary Burger`s equation:

$$\frac{1}{2}(U^2)_x = \mu U_{xx}$$

with a diffusion term on the right-hand side. The convective term on the left-hand side of the equation may be discretized at the mesh point xi by

$$(V_i^2)_x \doteq \frac{1}{\Delta x}(V^2_{i+\frac{1}{2}} - V^2_{i-\frac{1}{2}}) = \frac{1}{\Delta x}\left\{\left(\frac{V_{i+1}+V_i}{2}\right)^2 - \left(\frac{V_i+V_{i-1}}{2}\right)^2\right\}$$

so that the non-linear system of equations takes the form

$$f_i(V) = \frac{\alpha}{2}(V_{i+1}^2 + 2V_i(V_{i+1} - V_{i-1}) - V_{i-1}^2) - V_{i+1} + 2V_i - V_{i-1} = 0$$

for i = 2,3,...,n-1, V = (V1,V2,...,Vn) and alpha = delta x / 4µ 4µ. It can be shown that the Jacobian of the system is a diagonally dominant L-matrix (see e.g. [99]. The positive diagonals of the Jacobian are computed to be:

$$d_i(V) = 2 + \alpha(V_{i+1} - V_{i-1})$$

The iterative scheme then becomes:

$$\begin{aligned} V_i^{(k+1)} &= g(V_{i-1}^{(k+1)}, V_i^{(k)}, V_{i+1}^{(k)}) \\ &= V_i^{(k)} - \left\{\frac{\alpha}{2}\left((V_{i+1}^{(k)})^2 + 2V_i^{(k)}(V_{i+1}^{(k)} - V_{i-1}^{(k+1)}) - (V_{i-1}^{(k+1)})^2\right)\right. \\ &\quad \left. - V_{i+1}^{(k)} + 2V_i^{(k)} - V_{i-1}^{(k+1)}\right\} \Big/ \left(2+\alpha(V_{i+1}^{(k)} - V_{i-1}^{(k+1)})\right) \end{aligned}$$

for i = 2,3,...,n-1. In this form the scheme is highly recursive owing to the dependency between Vi(k+1) and V(i-1)(k+1). However, a vectorization is possible when applying the red-black re-ordering of section 5.4. In detail, we have two stages:

$$V_i^{(k+\frac{1}{2})} = g(V_{i-1}^{(k)}, V_i^{(k-\frac{1}{2})}, V_{i+1}^{(k)})$$

$$V_i^{(k+1)} = g(V_i^{(k+\frac{1}{2})}, V_{i+1}^{(k)}, V_{i+2}^{(k+\frac{1}{2})})$$

for i = 2,4,...,n-2 and n even, which may be evaluated independently. At each stage, the values of the right-hand side are known.

Under certain assumptions which are similar to those for linear, red-black ordering, convergence may be proven. However, the author has no knowledge whether or not the profitable monotone convergence property will be preserved.

5.8 RUNGE-KUTTA TIME-STEPPING METHODS FOR SYSTEMS OF ORDINARY DIFFERENTIAL EQUATIONS

Runge-Kutta methods have a unique place among the classical discrete variable methods for solving the initial value problem of ordinary differential equations

$$\frac{dy}{dx} = y' = f(x,y), \quad x \in [a,b]$$
$$y(a) = y_0$$

where y = (y1(x), y2(x),...,ys(x)) , and the initial value y0 is given . The idea is to construct a formula

$$y_{n+1} = y_n + h\phi(x_n, y_n, h), \quad n = 0,1,2,\ldots$$

which agrees with the Taylor series expansion of y as closely as possible without involving derivatives of f . An explicit formula of second-order accuracy which requires two evaluations of f per step is (for s = 1):

$$y_{n+1} = y_n + \frac{h}{2}\{f(x_n,y_n) + f(x_n+h, y_n+hf(x_n,y_n))\} .$$

The form of the methods is ideal for practical computations. They do not require any special starting procedure, in contrast to multistep methods such as Adams-Bashford method and allow easy change of stepsize. The most well-known formula is perhaps the 4-stage fourth order accurate Runge-Kutta method which, for s = 1, takes the following form:

$$y_{n+1} = y_n + \frac{1}{6}(k_1+2k_2+2k_3+k_4)$$

where:

$$k_1 = hf(x_n,y_n), \quad k_2 = hf(x_n+\frac{h}{2}, y_n+\frac{k_1}{2})$$

$$k_3 = hf(x_n+\frac{h}{2}, y_n+\frac{k_2}{2}), \quad k_4 = hf(x_n+h, y_n+k_3) .$$

For s = 1 the method is highly recursive and therefore not vectorizable. For s > 1, however, each Runge-Kutta stage may be evaluated simultaneously for y1, y2,...,ys. The larger the number s of unknowns, the faster the vector algorithm.

The application of the method to partial differential equations is now straightforward and leads to a large system of ordinary differential equations. By way of an example, we discuss the implementation of the classical 4-stage Runge-Kutta scheme for the model problem:

$$U_t = U_{xx}, \quad x \in [0,1], \quad t > 0$$

$$U(0,t) = U_0(t), \quad U(1,t) = U_1(t)$$

$$U(x,0) = g(x)$$

on vector computers. A method that has been studied extensively, is the "method of lines", also called semi-discrete approximation. The partial differential equation is reduced to a system of ordinary differential equations by simply replacing the spatial derivative of the right-hand side by finite differences. Let:

$$PV = \frac{1}{\Delta x^2}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} V$$

be the discretization of the right-hand side of the differential equation in the discrete interval [0,1] with step-size delta x and V = (V1,V2,...,Vs). The system of equations then results in the so-called semi-discrete representation:

$$\frac{dV}{dt} + PV = 0 .$$

The 4-stage Runge-Kutta scheme for the solution of this equation on time level n then becomes:

$$W^{(0)} = V^{(n)}$$
$$W^{(1)} = W^{(0)} - \frac{1}{2}\Delta t\, PW^{(0)}$$
$$W^{(2)} = W^{(0)} - \frac{1}{2}\Delta t\, PW^{(1)}$$
$$W^{(3)} = W^{(0)} - \Delta t\, PW^{(2)}$$
$$W^{(4)} = W^{(0)} - \frac{1}{6}\Delta t(PW^{(0)} + 2PW^{(1)} + 2PW^{(2)} + PW^{(3)})$$
$$= \frac{1}{3} (W^{(1)} + 2W^{(2)} + W^{(3)} - W^{(0)} - \frac{\Delta t}{2} PW^{(3)})$$
$$V^{(n+1)} = W^{(4)} .$$

In ([52],chapter 6.8), the scheme has been used to integrate the hyperbolic Euler equations for inviscid flow in time. There another multistage, two-level scheme of the Runge-Kutta type is revived, the so-called Gary scheme (1964). Again let n denote the time level (t = n*delta t), we have:

$$W^{(0)} = V^{(n)}$$
$$W^{(1)} = W^{(0)} - \Delta t\, PW^{(0)}$$
$$W^{(2)} = W^{(0)} - \frac{1}{2}\Delta t (PW^{(0)} + PW^{(1)})$$
$$W^{(3)} = W^{(0)} - \frac{1}{2}\Delta t (PW^{(0)} + PW^{(2)})$$
$$V^{(n+1)} = W^{(3)} .$$

It can be regarded as a Crank-Nicolson scheme with a fixed point iteration which terminates after three iterations. It is only second order accurate in time, but has the advantage that it needs less storage and offers more temporal damping (see [101], chapter 6.8) than the 4-stage method presented above.

Vectorization of this type of explicit methods is elementary and has been carried out for 3D aerodynamic simulation by many authors. All the vectors on the right-hand sides are known when needed. Evaluation of the 4-stage method leads one to the incomplete Taylor series expansion:

$$W^{(4)} = W^{(0)} - \Delta t\, PW^{(0)} + \frac{1}{2}\Delta t^2 P^2 W^{(0)} - \frac{1}{6}\Delta t^3 P^3 W^{(0)} + \frac{1}{24}\Delta t^4 P^4 W^{(0)} .$$

With the aid of Horner s notation for polynomial evaluation and Q = delta(t) * P we get:

$$W^{(4)} = \left(I - Q\left(I - Q\left(\frac{1}{2} I - Q\left(\frac{1}{6} I - \frac{1}{24} Q\right)\right)\right)\right) W^{(0)}$$

where I is the identity matrix. This involves four equations

$$W = W^{(n)}$$

$$W = (I - \frac{1}{4}Q)W$$

$$W = (I - \frac{1}{3}Q)W$$

$$W = (I - \frac{1}{2}Q)W$$

$$W = (I - Q)W$$

$$W^{(n+1)} = W \ .$$

These linked triads of the form

vector + scalar * (vector + scalar * vector)

in components, for our example,

$$W_i = W_i + \frac{\Delta t}{4h^2} * ((W_{i-1} + W_{i+1}) - 2.* W_i)$$

increase vector performance remarkably. One further advantage is the reduction in storage requirements which is only one quarter of that of the original 4-stage scheme. In FORTRAN, the vectorized scheme, without concern for boundary conditions, may have the following simple form (with IES1 = s-1 and DTH = delta(t) /delta(x)**2

```
   DO   1   K = 1,4
   F = DTH/FLOAT(5-K)
   DO   1   I = 2,IES1
 1 W(I) = W(I)+F*(W(I-1)+W(I+1)-2.*W(I))
```

5.9 SOME REMARKS ON EXPLICIT SCHEMES

Some further notes concerning explicit finite difference and finite volume methods should be made. Explicit schemes have some remarkable properties. Besides the better representation of the physical behavior of waves with finite speed, they imply low storage, easy programming and excellent time-accuracy for transient problems. Moreover, they have the advantage of being fully vectorizable, because there are no recursive solution processes and the length of the vectors can be made as long as the number of computational points or cells. They are easy to divide into subdomains and, therefore, easy to multitask.

One disadvantage for solutions which vary slowly in time and for steady state calculations, however, is the restriction of the time step due to restrictive stability requirements, which results in a much lower convergence rate than for implicit schemes. It is, therefore, necessary to relax this restriction and improve the convergence of explicit schemes by, e.g.

* Optimal Runge-Kutta stability bounds
* Enthalpy damping
* Local time-stepping and domain-splitting techniques which allow advancing the solution with different time steps at different grid points, cells or regions of the mesh
* Higher order spatial discretization, in which case the number of grid points may be reduced (if the solution is smooth enough) and thereby the time step increased
* Superstep acceleration techniques for parabolic type problems, the time steps being calculated from the reciprocals of the zeros of certain damped Chebyshev polynomials, [40],
* Explicit or implicit averaging of the residuals
* Multigrid or simple multiple grid techniques

Unfortunately, some of these recommendations are not very efficient for the solution of the turbulence-averaged Navier-Stokes equations, so that implicit methods, today, have yet some advantages over explicit ones (see chapter 6).

5.10 LITERATURE

[1] Alefeld G.: On the convergence of the symmetric SOR method for matrices with red-black ordering. Numer. Math. 39 1982, 113-117.

[2] Ames W.G.: Sparse matrix and other high performance algorithms for the CRAY-1. Systems Engineering Lab. Univ. Michigan, Report 124, 1979.

[3] Anderson D.V.: Avoiding Common Errors of Multitasking. Several articles in NMFECC Buffer 1986.

[4] Bailey D.M.: A fast Fourier Transform Without Power-of-Two Memory Strides. NASA Ames 1986.

[5] Barlow R.H., Evans D.J., Shanehchi J.: Parallel multisection applied to the symmetric tridiagonal eigenvalue problem. Computer J. 25 1982.

[6] Barlow R.H., Evans D.J., Shanehchi J.: Sparse matrix vec tor multiplication on the ICL-DAP. Proc. Conf. on Progress in the Use of Vector and Array Processors 1983.

[7] Brown F.: A high performance scalar tridiagonal equation solver for the CRAY-1. Dept. Nuclear Engineering, Univ. Michigan 1980.

[8] Bunemann O.: Complex Arithmetic on the CRAY-2. NMFECC Buffer, Sept. 1986.

[9] Bunemann O.: Vector FFT for the CRAY-2. NMFECC Buffer, Nov. 1986.

[10] Calahan D.A.: A Block-Oriented Sparse Equation solver for the CRAY-1. Int. Conf. Paral. Proc. Bellaire, MI 1979, pp 234-239.

[11] Calahan D.A., Ames W.G., Sesek E.J.: A collection of equations solving codes for the CRAY-1.Systems Engineering Lab. Univ. Michigan, Report No. 133, 1979.

[12] Calahan D.A.: Sparse vectorized direct solution of elliptic problems. In: Elliptic problem solvers (Schultz, M., ed.), Acad. Press, New York 1981, 241-245.

[13] Calahan D.A.: A vectorized general sparsity solver. Systems Engineering Lab. Univ. Michigan, Report 168, 1982.

[14] Calahan D.A.: High-performance banded and profile equation solver for the CRAY-1. I. The symmetric case. Systems Engineering Lab. Univ. Michigan, Report 160, 1982.

[15] Calahan D.A.: High-performance banded equation solver for the CRAY-1. II. The symmetric case. Systems Engineering Lab. Univ. Michigan, Report 166, 1982.

[16] Calahan D.A.: Task Granularity Studies on a Many-Processor CRAY X-MP. Parallel Computing 2, 1985, 109-118.

[17] Calahan D.A.: Block-Oriented Local-Memory-Based Linear Equation Solutions on the CRAY-2: Uni-processor Algo rithms. 1986.

[18] Chen S.C., Kuck D.J., Sameh A.H.: Practical parallel band triangular system solvers. ACM Trans. Math. Software 4 1978, 270-277.

[19] Curtis B.: Local Memory and CRAY-2 Performance. NMFECC Buffer, Sept. 1986.

[20] Dave A., K. and Duff, I.S.: Sparse Matrix Calculations on the CRAY-2. Conf. Leon Norway, June 1986.

[21] Dongarra J.J., Hinds A.R.: Unrolling loops in Fortran. Software-Practice and Experience 9, 1979, 219-229.

[22] Dongarra J. J., Eisenstat S.C.: Squeezing the Most out of an Algorithm in CRAY Fortran. Argonne Nat. Lab., TM No.9, 1983.

[23] Dongarra J. J.: Redesigning linear algebra algorithms. Proc. 1 Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983, 51-59.

[24] Dongarra J.J. et al.: Implementing Linear Algebra Algo rithms for Dense Matrices on a Vector Pipeline Machine. SIAM Review. 26, 1984, pp 91-112.

[25] Dongarra J.J. and Duff I.S.: Performance of vector com puters for direct and indirect addressing in Fortran. Har well Report 1986.

[26] Dongarra J.J., Kaufmann, L., and Hammarling, S.: Sgeezing the most out of eigenvalue solvers on high-performance computers. Lin. Alg. Appl. 77, 1986, 113-136.

[27] Dongarra J.J., Sameh, A.H., Sorensen, D.C.: Implementation of some Concurrent Algorithms for Matrix Factorization, Parallel Computing 3, 1986, 25-34.

[28] Dubois P.F., Rodrigue G.H.: Operator splitting on vector processors. Lawrence Livermore Lab., UCRL-79316, 1977.

[29] Duff I.S.: The solution of sparse linear equations on the CRAY-1. Proc. NATO Advanced Research Workshop on High- Speed Computation Juelich, 20-22 June, 1983.

[30] Duff I.S., Erisman A.M., and Reid J.K.: Direct methods for sparse matrices. Oxford University Press, London 1986.

[31] Feilmeier M. (ed.): Parallel computers - parallel mathema tics. North-Holland, Amsterda, 1977.

[32] Feilmeier M., Roensch W.: Parallel nonlinear algorithms. Computer Physics Comm. 26 1982, 335-348.

[33] Feilmeier M., Joubert G., Schendel U. (eds.): Proceedings of "Parallel Computing 83", North-Holland Publ.1984.

[34] Fong K., Jordan T.L.: Some linear algebraic algorithms and their performance on the CRAY-1. Los Alamos Scient. Lab., Report LA-6774, 1977.

[35] Fong K.: Several articles on multitasking on the CRAY-2 in the NMFECC Buffer 1985/86.

[36] Fornberg B.: A vector implementation of the Fast Fourier Transform algorithm. Math. Comp. 36 1981, 189-191.

[37] Gentzsch W.: Recursion algorithms on vector computers. Proc. 1st Int. Conf. on Vector and Parallel Computing in Scientific Applications, Paris 1983, 79-86.

[38] Gentzsch W.: A Survey on Finite Volume Methods. DFVLR-IB 221-84 A 10, 1984.

[39] Gentzsch W., Schaefer G.: Solution of large linear systems on vector computers. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[40] Gentzsch W.: Vectorization of Computer Programs with Applications to Computational Fluid Dynamics. Vieweg Publ. Comp., Braunschweig (F.R.G.) 1984.

[41] Gentzsch W.: The Optimal Use of Vector Computers in Com putational Physics. Proc. SEAS 1985.

[42] Gentzsch W.: Vectorization of Numerical Algorithms Demon strated for the SOR Method. Regensburg Dec. 1986.

[43] Gentzsch W.: A Fully Vectorizable SOR Variant. Parallel Computing, 1987.

[44] Gentzsch W., Neves, K.: Memories in Supercomputers. Proc. SEAS Montpellier 1987.

[45] Greenbaum A., Rodrigue G.: The ICCG method for the STAR. Res. REP. UCID-17574, Lawrence Livermore Lab., 1977.

[46] Haendler W. (ed.): Proc. CONPAR 81. Lecture Notes in Computer Science, Vol. 111, Springer Berlin 1981.

[47] Heller D.E.: Some aspects of the cyclic reduction algo rithm for block tridiagonal linear systems. SIAM J. Numer Anal., 13 1976, 484-496.

[48] Heller D.: A survey of parallel algorithms in numerical linear algebra. SIAM Review 20 1978, 740-777.

[49] Hockney R.W., Jesshope C.R.: Parallel Computers- Architec ture, Programming and Algorithms. Adam Hilger, Bristol 1981.

[50] Johnson O.G., Paul G.: Vector algorithms for elliptic partial differential equations based on the Jacobi method. In: Elliptic Problem Solvers (Schultz, M., ed.), Acad. Press New York 1981, 345-351.

[51] Jordan T.L.: A new parallel algorithm for diagonally dominant tridiagonal matrices. Los Alamos Scientific Lab. Report 1974.

[52] Kascic M.J.: A direct poisson solver on STAR. CDC Minneapolis 1978.

[53] Kascic M.J.: Anatomy of a Poisson solver. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[54] Kershaw D.S.: The solution of single linear tridiagonal systems and vectorization of the ICCG algorithm on the CRAY-1. Res. Rep. UCID - 19085, Lawrence Livermore Lab., 1981.

[55] Kogge P.M.: Maximal rate pipeline solutions to recurrence problems. Proc. 1. Ann. Symp. on Comp. Architectures 1973, 71-76.

[56] Kogge P.M.: A parallel algorithm for the efficient solu tion of a general class of recurrence equations. IEEE Trans. Comp., C-22 1973, 786-793.

[57] Korn D.G., Lambiotte J.J.: Computing the Fast Fourier Transform on a vector computer. Math. Comp. 33 1979, 977- 992.

[58] Kuck D.J., Lawrie D.H., Sameh A.H.: High speed computer and algorithm organization. Acad. Press, New York 1977.

[59] Lambiotte J.J., Voigt R.G.: The solution of tridiagonal linear systems on the CDC STAR-100 computer. ACM Transac tions on Math. Software, 1 1975, 308-329.

[60] Lemke M.: Experiments with a Vectoized Multigrid Poisson Solver on the CDC Cyber 205, CRAY X-MP and Fujitsu VP 200. GMD Report Bonn (F.R.G.) 1985.

[61] Lichnewsky A.: Some vector and parallel implementations for preconditioned conjugate gradient algorithms. Proc. NATO Advanced Research Workshop on High-Speed Computation, Juelich, 20-22 June, 1983.

[62] Loehner R., Morgan, K.: Finite Element Methods on Super computers: The Scatter-Problem. Proc. NUMETA Conf. 1985.

[63] Luk C.: Memory Interleave on the CRAY-2. NMFECC Buffer, Sep. 1986.

[64] Madson N.K., Rodrigue G.H., Karush J.I.: Matrix multipli cation by diagonals on a vector/parallel processor. Info Processing Letter, 5 1976.

[65] Meier U.: A Parallel Partition Method for Solving Banded Systems of Linear Equations. Parallel Computing 2, 1985, 33-44.

[66] Natvig J., Nour-Omid B., Parlett B.N.: Effect of the CYBER 205 on the Choice of Method for Solving the Eigenvalue Problem. (A-sM)x = 0. J. Comp. Appl. Math. 15, 1986, 137- 159.

[67] Oed W., Lange O.: Transforming linear recurrence relations for vector processors. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[68] Oed W., Lange O.: On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems. IEEE Trans. Comp., C-34, 1985, 949-957.

[69] Oed W., Lange O.: Modelling, Measurements, and Simulation of Memory Interference in the CRAY X-MP. Parallel Computing 3, 1986, 343-358.

[70] O'Leary D.P.: Parallel Implementation of the Block Conjugate Gradient Algorithm. Parallel Computing 1987.

[71] Ortega J.M., Voigt R.G.: Solution of Partial Differential Equations on Vector and Parallel Computers. SIAM Rev. 27, 1987, 149-240.

[72] Radicati G., di Brozolo M., Vitaletti M.: Sparse Matrix- Vector Product and Storage Representations on the IBM 3090VF, ECSEC Report, Roma 1986.

[73] Reed D.A., Patrick M.L.: Parallel, Iterative Solution of Sparse Linear Systems: Models and Architectures. Parallel Computing 2, 1985, 45-68.

[74] Robert Y., Sguazzero P.: The LU Decomposition Algorithm and its Efficient Fortran Implementation on the IBM 3090 Vector Multiprocessor. ECSEC Report, Roma 1987.

[75] Rodrigue G.H., Madson N., Karush J.: Odd-even reduction for banded linear equations. Lawrence Livermore Lab., UCRL-78652, 1976.

[76] Rodrigue G.H.: Operator splitting on the STAR without transposing. Lawrence Livermore Lab., UCRL-17515, 1977.

[77] Rodrigue G.: Parallel Computations. Acad. Press, New York 1982.

[78] Sack R.A.: Relative pivoting for systems of linear equations. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[79] Sameh A.H., Chen S.C., Kuck D.J.: Parallel Poisson and biharmonic solvers. Computing 17 1976, 219-230.

[80] Sameh A.H., Brent R.P.: Solving triangular systems on a parallel computer. SIAM J. Numer. Anal. 14 1977, 1101- 1113.

[81] Sameh A.: Numerical parallel algorithms - A survey. In: High Speed Computer and

Algorithm Organization (Kuck, D.J., Lawrie, D.H., Sameh, A.H., eds.), Acad. Press, New York 1977, 207-228.

[82] Schmidt H., Schumann U., Volkert H.: 3D Diıect and Vectorized Elliptic Solvers for Various Boundary Conditions. DFVLR-Mitt. 84-15, 1984.

[83] Schnepf E., Schoenauer W.: Parallelization of PDE software for vector computers. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[84] Schoenauer W., Raith K.: A polyalgorithm with diagonal storing for the solution of very large indefinite linear banded systems on a vector computer. Proc. 10the IMACS World Congress on System Simulation and Scientific Computation 1 1982, 326-328.

[85] Schoenauer W.: The efficient solution of large linear systems, resulting from the FDM for 3-D PDE's, on vector computers. Proc. 1. Int. Coll. on Vector and Parallel C ıputing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Se.ie C, 1 1983, 135-142.

[86] Schoenauer W.: Numerical experiments with instationary Jacobi-OR methods for the iterative solution of linear equations. ZAMM 63 1983, T380-T382.

[87] Schoenauer W., Gentzsch W. (Eds.): The Efficient Use of Vectorcomputers with Emphasis on CFD. Vieweg Publ. F.R.G. 1986.

[88] Stone H.S.: An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. JACM, 20 1973, 27-38.

[89] Stone H.S.: Parallel tridiagonal solvers. Digital Systems Lab., Stanford Univ. 1974. And: J. ACM 20 1973, 27-38.

[90] Schwarztrauber P.N.: Vectorizing the FFT's. In: Parallel Computations (Rodrigue, G., ed.), Acad. Press, New York 1982.

[91] Temperton C.: Fast Fourier Transforms on the CYBER 205. Proc. NATO Advanced Research Workshop on High-Speed Computation, Juelich, 20-22 June, 1983.

[92] Temperton C.: Fast Fourier Transform for numerical prediction models on vector computers. Proc. 1. Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983, 159-162.

[93] Traub J.F. (ed.): Complexity of sequential and parallel numerical algorithms. Acad. Press New York 1977.

[94] van der Vorst H.A.: A Vectorizable Variant of some ICCG methods, SIAM J. Sci. Stat. Comput. 3, 1982, 350-356.

[95] van der Vorst H.A.: On the vectorization of some simple ICCG methods. Proc. 1. Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983.

[96] van der Vorst H.: The Performance of Fortran Implementations for Preconditioned Conjugate Gradients on Vector-Computers. Parallel Computing 3, 1986, 49-58.

[97] van der Vorst H.: Large Tridiagonal and Block Tridiagonal Linear Systems on Vector and Parallel Computers. Parallel Computing 1987.

[98] Voigt R.G.: The influence of vector computer architecture on numerical algorithms. ICASE Report 77-8, NASA Langley 1977.

[99] Young D.M.: Iterative Solution of Large Linear Systems. Academic Press, New York 1971.

CHAPTER 6: COMPUTATIONAL FLUID DYNAMICS AND SUPERCOMPUTERS (Prof. Dr. Gentzsch)

As is well-known it is most important to optimally adapt codes and algorithms to the vector or parallel computer in use. One conclusion is that in addition to faster and larger supercomputers, users must be much better trained than for (scalar) general purpose computers. Therefore, in the following, we present some details on restructuring typical numerical algorithms to achieve superior performance on vector computers. The focus, of course, is on Computational Fluid Dynamics.

During the last two decades computational fluid dynamics (CFD) gained an important position together with experiments in wind-tunnels and analytical methods. The main objective of CFD is to simulate dynamic flow fields through the numerical solution of the governing equations, e.g. the Navier-Stokes equations, using high-speed computers.

The simulation of two-dimensional (2D) inviscid and viscous flows on vector computers does not represent any difficulties with respect to memory requirements or computation time. In three dimensions (3D), however, one has to compute about 20 to 30 variables per mesh point in a 3D-field per time-step or iteration such as the velocity components, density, pressure, enthalpy, temperature, concentrations, dissipative fluxes, local time steps, geometry coefficients, dummy arrays etc. The computations in the case of 3D are therefore restricted to fairly coarse meshes as well as to solutions which are often not fully converged solutions. The large amount of CPU (Central Processing Unit) time involved and the fact that the data cannot be contained in central memory are the main reasons for the long elapsed times for CFD applications. In these cases, the mapping of the problem onto the architecture of the machine and in particular onto special organization of the memory must be carefully considered to take full advantage of the vector computer.

The in-core possibilities of a vector computer with one million words main memory are fairly restricted, e.g. (depending on the method)

less than 40 x 30 x 20 grid points for Navier-Stokes
less than 50 x 30 x 30 grid points for Euler
less than 160 x 30 x 30 grid points for grid generation.

However, a "Navier-Stokes" mesh must be very fine, say one million grid points, to better understand the physical flow phenomena, to obtain quantitative results of forces, and to study dynamic behavior depending on different parameters. The same is true for the Euler equations, except that viscous flow is not simulated.

During a Workshop [107] some examples of very large problems have been presented for the Euler solutions on

* 3 x 10E+5 grid points for the DFVLR F4 wing
* 1 x 10E+6 grid points for the Dillner wing
* 2.5 x 10E+6 grid points for the Dillner wing;

for the Euler and Navier-Stokes solutions on

* 1 x 10.5 grid points for hemisphere cylinder;

and for the Navier-Stokes solutions on

* 0.9 x 10E+5 grid points for hemisphere cylinder
* 2.1 x 10E+5 grid points for hemisphere cylinder.

Some authors presented several dense-mesh solutions with much greater details e.g. in the vortex-shock-wave structure or the separation region for the fine mesh solutions.

The largest problem with 2.5 x 10E+6 grid points has been performed on a CRAY X-MP/48 vector computer with a 128 million word Solid-state Storage Device (SSD). This mesh required approximately 7 million words of internal memory and 100 million words of external memory. The CPU time for convergence at 800 iterations was 7.9 hours, with a total elapsed time of 8.3 hours.

The 1 x 10E+6 grid problem has been performed on a CYBER 205 with 16 million 64-bit words main memory. The data set was 23 M 32-bit words. The CPU time here for 1000 iterations was 2.5 hours with 2.5 hours elapsed time.

From these examples it is obvious that only the fastest supercomputers, namely vector and parallel computers, with large primary and secondary memories, enable us to solve large problems in a reasonable time.

6.1 SUMMARY OF A WORKSHOP ON CFD AND VECTOR COMPUTERS

In 1985 a workshop on "the efficient use of vector computers with emphasis on CFD" was organized at the University of Karlsruhe (F.R.G.), and in 1986 the corresponding proceedings with 18 papers on vectorization and multitasking of CFD codes were published [107]. The problems treated by the participants cover the wide area of CFD and applications so that a brief summary of the main topics would be very suitable and informative for the interested reader. A more general view on advances in CFD then will follow in the next section.

MAIN BOTTLENECKS OF VECTOR COMPUTERS

An important point discussed in detail are the bottlenecks arising in vector computers, which are the main reasons for (sometimes sophisticated) restructuring of today's CFD codes. The early machines, such as CRAY-1 and CYBER 205, had severe bottlenecks which, nowadays, have been improved or removed in some machines such as the CRAY X-MP, the UNISYS ISP and the NEC SX-2 (see chapter 2). To remember, the main bottlenecks for the CRAY-1M/1S were:

* only one memory access port from main memory to vector registers which results in one word transfer per cycle (as with the CRAY-2)
* small main memory
* memory bank conflicts (for noncontiguous vector of constant stride)
* no gather/scatter hardware instructions
* excessive I/O for out-of-core programs;

and for the CYBER 205:

* max. vector length 65535
* vectors have to be contiguous in memory
* long start-up for short vectors
* no fast secondary storage
* excessive paging for large problems
* poor autovectorizer.

Knowledge of these shortcomings is very important for the CFD user to carry out code vectorization in an optimal way.

One drawback of most of today's supercomputers is the partitioning of the memory into memory banks (chapter 2, see also [44] in chapter 5.10).Other additional bottlenecks may arise for the (few and short) registers and the paths to memory as described in chapter 2 in detail.

To illustrate the decrease of performance due to memory bank conflicts, we present the performance (in MFLOPS) for the VP-200 depending on the stride of the vectors for different simple arithmetic operations for two vector lengths. The VP-200 configuration under consideration has 128 banks in 16 segments, 8 banks each. A similar behavior is valid for nearly all today's vector computers with memory banks.

TABLE 6.1: VP-200 performance in MFLOPS for 'z=x+y'

Stride	MFLOPS (N=1000)	MFLOPS (N=60)
1	114	24
2	45	23
3	85	24
4	23	16
5	85	24
8	21	16
16	16	13
32	8	8
64	4	4

TABLE 6.2: VP-200 performance in MFLOPS for 's=s+x*y'

Stride	MFLOPS (N=1000)	MFLOPS (N=60)
1	231	48
2	90	47
3	170	48
4	47	36
5	167	48
8	43	33
16	31	26
32	17	16
64	8	8

TABLE 6.3: VP-200 performance in MFLOPS for 'z=(x-y)*(x+y)'

Stride	MFLOPS (N=1000)	MFLOPS (N=60)
1	239	59
2	125	53
3	227	56
4	64	46
5	226	57
8	59	45
16	44	34
32	23	21
64	12	11

BASIC EQUATIONS

The main problem in CFD is to solve the basic system of nonlinear differential equations, called the Navier-Stokes equations - consisting of the continuity, momentum and energy equations - by numerical methods on high speed computers. Because of the complicated nature of these equations, simpler models have also been derived and solved numerically. (Thin layer Navier/Stokes equations are given later in Section 6.4.)

While the full potential equation models inviscid irrotational flows, the Euler equations describe inviscid flows with rotation. On the other hand, viscous incompressible and compressible, laminar and turbulent flows with e.g. separation are best characterized by the Navier-Stokes equations. In the following, we give a list of the different governing equations solved by the participants of the above mentioned workshop:

* 2D/3D Poisson equation for different purposes (grid generation, pressure iteration, Stokes problem)
* 3D full potential equation
* 2D Euler equations
* 3D Euler equations
* 2D Navier-Stokes equations in primitive variable formulation
* 2D Navier-Stokes equations in velocity-vorticity formulation
* 2D Navier-Stokes equations in vorticity-stream-function formulation
* 2D shallow-water equations
* 3D Navier-Stokes equations in thin-layer formulation
* 3D Navier-Stokes equations (Stokes-problem with incompressible convection equation),
* 3D Navier-Stokes equations in velocity-vorticity formulation
* 3D Navier-Stokes equations in primitive variable formulation

BOTTLENECKS CAUSED BY THE EQUATIONS AND THE PHYSICS

Because of the non-linearity of the equations, the boundary conditions, the discontinuities, the complex geometries, the need for time-accurate results for unsteady phenomena, etc., the numerical solution on vector computers becomes very difficult. Some important drawbacks arising in the various equations are as follows:

for the full potential equation-

* type-dependent discretization for subsonic/supersonic mesh points.

for the Euler equations-

* time-dependent calculation for the steady state of transonic flows resulting in long-time computations
* non-physical (extrapolation) boundary conditions at walls, outflow, and farfield
* need for artificial viscosity or one-sided differences (of lower order)
* strong gradients near shocks

for the Navier-Stokes equations-

* those mentioned for the Euler equations and, additionally,
* involve second order derivative terms
* thin boundary layer for high Reynolds number flows (stiffness)
* need for high resolutions in regions with strong gradients, separation, transition, etc.
* turbulence modeling
* partly non-physical (extrapolation) boundary conditions
* div U = 0 for incompressible flows and pressure correction

NUMERICAL ALGORITHMS

One possible classification of vector algorithms has been proposed in [107] dividing the algorithms into the following three subgroups:

* unchanged general purpose computer algorithms with many recursions, short vector length, which have not been designed for vectorization
* "vectorized" general purpose computer algorithms which solve the 2D or explicit 3D problems very fast, but are not suited for large data sets
* "vectorized" algorithms for large implicit 3D problems, with optimal data structure for out-of-core and performance not limited by the speed of the vector pipes but by the I/O bottleneck of most of the vector computers

Various numerical algorithms belonging to these groups have been implemented on vector computers. A detailed description of the vectorization of most of the following algorithms may be found in the individual contributions of [107] and in other papers mentioned in the list of references. We therefore restrict ourselves to listing the corresponding algorithms:

* Conjugate gradient methods and variants, e.g. ICCG, BI-CG
* FIDISOL - a finite difference solver
* successive line overrelaxation
* LU - decomposition
* LL - decomposition
* finite volume spatial discretization
* Runge-Kutta time stepping
* MacCormack schemes
* van Leer scheme
* Approximate tri-diagonal factorization of Beam and Warming and bi-diagonal factorization of MacCormack
* FLO 22
* FLO 57
* Osher's Riemann solver
* finite elements
* fourth order space approximation
* alternating direction implicit
* operator compact implicit
* Thomas algorithm
* Uzawa algorithm for Stokes problem
* Chorin's method for pressure correction
* Frontal method for linear systems
* TEACH code
* Fast Fourier transformation
* Poisson solver POISSX, POISSV

A strategy of vectorization together with many suggestions on how to overcome the main drawbacks caused by algorithms have been given in the previous chapter. Additionally in the second part of this chapter vectorization for several CFD algorithms is explained in more detail.

FLUID DYNAMIC PROBLEMS ON VECTOR COMPUTERS

A list of fluid dynamic problems treated during the workshop, for different vector computers, is taken from [107]:

* 2D heat driven cavity
* 3D convective flow
* 3D full potential code for transonic flow past wings
* 3D grid generation method
* Transonic flow past ONERA M6 wing using Euler equations
* Transonic flow past Dillner wing
* Leading-edge vortex flow around Dillner wing
* Flow past airfoils
* 2D shallow water flow problems
* 2D shock boundary layer interaction
* 2D flow past a cylinder
* Transonic/supersonic channel flow
* 3D transonic flow past a hemisphere-cylinder
* 3D flow around cylinder
* Cooling tower outfall problem
* Laser fusion problem
* Taylor-green vortex
* Atmospheric flows with long time scales (climate models)

INPUT/OUTPUT TRANSFER

Another problem discussed widely in the literature is the excessive input/output transfer between CPU and (secondary) out-of-core storage, which is due to the limited amount of central memory and/or the extreme data size arising mainly in the computation of 3D problems.

3D Euler solutions, for example, on meshes containing about 50,000 grid points and 3D Navier-Stokes solutions on meshes containing about 20,000 grid points, can be obtained on a machine with one million words of main memory without using extended storage on disk. These problems are also very suitable for today's mini-supercomputers. Further mesh refinement, however, is often necessary in order to validate the flow model and to better understand the physics of the resulting numerical solutions. Therefore, for almost any 3D application, the amount of data to be handled exceeds the core memory size except on large memory machines. Only the active data can be kept in central memory while all the remaining data have to be stored on out-of-core storage devices, e.g. on discs. This results in rather long I/O waiting times. In order to avoid, to reduce or to speed up I/O transfer for these data during the computations, the following proposals may be found in the literature:

* Split the computational grid into a few blocks of grid points, often called zones, substructures, subdomains or simply "blocks". All variables are then stored in arrays with length equal to the number of points in one zone. Two examples are the pencil concept and the plane concept. The pencil concept is used to arrive at long vector strings. In the plane concept a certain number of planes are contained in the core of the computer, such that all terms needed in the governing equations can readily be evaluated.

* Separate the optimal data selection from the processing of the data, and develop a data flow algorithm to optimally handle data blocks in actual use for the program, e.g. by dynamic data management routines. Strategies used to achieve efficient I/O operations in this context are random direct access file, unblocked buffered I/O, file banking, i.e. parallel I/O with several disk channels.

* Use of several time step calculations in each block before changing to the next block.

* In some cases, I/O may even be avoided by recomputing some quantities at each iteration (e.g. quantities depending only on the mesh and not on the solution field) rather than to read them from an auxiliary file after an initial computation. However, the storage of the solution, and the necessary number of iterations or time levels, depending on the scheme used, limit the possible reduction of the I/O.

* Perform 32-bit words calculations on CYBER 205 and UNIVAC ISP as often as possible and 64-bit words only where necessary. This not only results in a further reduction of I/O but also speeds up the CPU processing time.

* Use faster out-of-core storage devices or vector machines with a much larger main memory [107]. The CRAY Solid-state Storage Device, for example, transfers data at rates up to 100 times faster than disk drives. The use of the 128 million word SSD for a problem with 2.5 million grid points reduced the I/O transfer waiting time remarkably.

NEAR FUTURE

A conclusion resulting from the contributions of the workshop and some extrapolations for the foreseeable future have been made by the participants. Besides improvements in

* CPU processing time and
* much larger central memories

for current generation vector computers (such as the CRAY-2 with 4ns cycle time, 4 processors, up to 256 MWords central memory and up to 2 GIGAFLOPS, or the ETA 10/E with about 8ns cycle time, up to 8 processors and up to about 3 GIGAFLOPS) the computational fluid dynamicists expect further improvements in vector computer software. Progress is expected in

* FORTRAN 8X, which will be much better suited for vector computers

* Autovectorizers, which should vectorize many of the items in the earlier mentioned "checklist" (section 4.5)

* "Automultitaskers", which automatically distribute the work equally among the different processors (such as for the Alliant FX/8)

* Capability of managing problems with complex geometries or even unstructured grids which implies handling of irregular data structures

* More detailed diagnostics to better support vectorization, multitasking and I/O.

On the other hand, much greater are the improvements to be made by mathematicians, physicists and engineers in improving and developing new numerical algorithms which

* are more accurate, therefore requiring fewer grid points and less storage requirements,

* are better suited for vector and parallel processors, and

* have a much faster convergence rate than today's algorithms.

Finally, a very close cooperation of the CFD users and the vector computer manufacturers is of growing importance.

6.2 ADVANCES IN COMPUTATIONAL FLUID DYNAMICS (CFD)

It is nearly impossible to present a complete survey of the advances in CFD as an aerospace-system design tool because they are proceeding at such a rapid pace that their capability is not widely known by the aerospace community. Fortunately, in 1985 a committee has been established by the U.S. Governing Board of the National Research Council to assess current capabilities and future directions of CFD. Based on a nation-wide survey a study has been published in 1986 [26] reporting on significant findings and specific recommendations for future directions for three application areas - external aerodynamics, hypersonics and propulsions - followed by a turbulence modeling discussion. The author believes that a synopsis of these subjects with special emphasis on the numerical solution of the Navier-Stokes equations fits the purpose of this AGARD-o-GRAPH in an ideal way.

CFD is becoming an increasing powerful tool in aerodynamic design of aerospace systems as a result of improvements in numerical algorithms as well as in the processing speed and storage capacity of new generations of computers. As the next generation of super-computers becomes available much current CFD work may be expanded to address more complex configurations, geometries, flight regimes, flow fields, and applications, while some of the existing work will become components of more complex systems of solutions to problems of modeling, code generation, flow field solvers, and flow visualization. Current CFD methods have only demonstrated an ability to simulate flows about complex geometries with simple physics or about simple geometries with more complex physics. In general, they cannot simulate flows about complex geometries with complex physics.

An increasing number of issues related to computer science are appearing in the development of advanced computational capabilities. As nonlinear codes begin to treat more complex 3-D configurations, the issues of

* vectorization
* multitasking
* large data bases
* analysis of computed results
* 3-D graphic displays
* surface and grid generation

become increasingly important. In the future stronger emphasis will be on the development of more advanced algorithm technology, particularly for the Euler equations and the various forms of the turbulence-averaged Navier-Stokes equations. One objective is technology to enable computation of flows about arbitrary and complex geometries in a practical and accuracy predictable manner. Another objective is for major improvements in spatial accuracy and convergence. With that, order-of-magnitude improvements are possible. In the future the overall progress in CFD will depend primarily on four factors [26]:

* the power and storage capacity of the computers available

* the ability to generate mesh systems for complex configuration

* the construction of algorithms for the solution of the flow field equations which can deal with discontinuities arising from shock waves and vortex sheets

* the capability for modeling turbulent flows

EXAMPLES OF CFD IMPACT ON AIRCRAFT DESIGN

An extensive review of the development and outlook for computational aerodynamics up to 1979 was given by Chapman [24] in his Dryden lecture. Another state of the art report on large-scale computing being a very important contribution to the same subject was the AGARD report no. 209 in 1984 [73]. However, in the context of the present study, three years after, it is felt that a few more modern examples of the application of numerical methods to aircraft design problems seem to be appropriate.

The first example concerns the design of a nacelle installation taken from Rubbert [105]. For conventional underwing-mounted turbofan engines, it is desirable for the vertical location of the nacelle to be close to the wing. This is particularly important in order to minimize the landing gear length (and hence weight) for safe ground clearance with the modern high bypass engines. Designers found that if a nacelle was positioned too close to the wing, the drag increased to unacceptable levels. The source of this unwanted drag was not made clear by wind tunnel testing, but designers coined a name for the mysterious quantity. They called it "interference drag".

A computational attack aimed at understanding and resolving this problem was initiated several years ago. The first step was to calibrate and gain confidence in a computational modeling of the wing/strut/nacelle/plume flow, which took some time and learning. But with that in hand, three different nacelle installations were analyzed and compared with experiment, and good correlation between computed and measured drag was obtained. The important point is that the computation revealed the source of the interference drag, which the wind tunnel had been unable to do. It was none other than induced or vortex drag caused by a change in wing span loading due to the presence of the nacelle and strut. (One asset of computational methods is that it is straightforward to individually calculate the induced drag, wave drag, and profile drag. The usual wind tunnel test provides only the total drag).

With that information in hand, the design solution became clear; namely to contour the nacelle and strut to prevent adverse impact to the spanwise load distribution of the wing. Properly contoured nacelle and strut design was done much more effectively with computational methods than with the wind tunnel because computational methods automatically produce finely detailed pressure distributions on all wing, strut, and nacelle surfaces. In the process, an added bonus was achieved by carefully redesigning the section shape of the strut to reduce local supervelocity levels that were contributing unnecessarily to profile drag and that could lead to local shock wave formation at the higher Mach numbers.

The knowledge and computational experience thus obtained subsequently were applied to the design of the Boeing 757, 767, 737-300, and KC-135R nacelle installations, enabling very close-coupled installations to be achieved without incurring a significant drag penalty. In this instance, computations allowed the achievement of a configuration goal that was never achieved by 20 years of wind tunnel testing and experimentation [105].

The second example demonstrates the capability of CFD for the solution of a high-lift problem [105]. A puzzling and potentially serious problem arose during flight test of a Boeing 707 re-engined with larger diameter CFM 56 engines. The problem was an unexpected loss of 10 percent in maximum lift capability that appeared during flight test but had not been predicted by wind tunnel tests. At first, there appeared to be no obvious experimentally derivable aerodynamic "fix-up" short of extensive full-scale flight experimentation that would be prohibitively expensive.

Subsequent flow visualization work showed that at wind tunnel Reynolds numbers, the maximum-lift characteristics of the wing were dominated by the outboard section characteristics. At flight Reynolds numbers, the outboard wing sections benefited from the increased Reynolds number, and the maximum-lift performance became limited by an unfavorable inboard wing boundary layer/nacelle vortex interaction. Thus, different physical mechanisms were dominating the maximum-lift characteristics at wind tunnel and flight conditions.

Hence, the puzzle was solved, but the problem was not. The traditional approach would be to embark on an expensive and time-consuming flight test fix-up program. However, with the availability of computational tools, a quite different approach became feasible. The approach was to find a way to simulate the full-scale aerodynamics, rather than the full-scale geometry, in the wind tunnel. This required the use of computational tools with design (inverse) capability.

It was a straightforward procedure with computational tools to design an outboard leading edge device to forestall outboard separation at wind tunnel Reynolds numbers. This device was fitted to the outboard wing of the wind tunnel model. (Note: at no time was it intended to fit such an alternative leading-edge device to the full-scale wing). In this way, the outboard wing behaved at wind tunnel Reynolds number very much like the full-scale wing in flight; that is, nacelle vortex/wing boundary-layer interactions now determined the stall phenomena in the wind tunnel.

Having now radically adjusted the wing's stall patterns in the wind tunnel, attention could turn to possible modifications to improve the maximum-lift performance. A simple fix in the form of a nacelle-mounted vortex control device was found that delayed the stall of the inboard wing associated with the nacelle vortex phenomenon. .Thus, no change to the baseline high-lift system of the full-scale airplane was required, and the baseline flight level maximum-lift performance was fully regained.

In this case, computational methods provided an effective approach to solving a problem that could not have been done solely through wind tunnel testing, and it is a very lucid example of the complementary relationship between the wind tunnel and CFD [105].

CODES FOR AERODYNAMIC FLOWS

(Linear) panel codes and (nonlinear) small-disturbance codes nowadays are in regular use within every major aerospace company. The computer requirements for these codes are modest - less than one-half million words of memory and a few minutes of CPU-time on a Class VI computer are required for simple wing-body combinations. Therefore a careful and time-consuming hand-vectorization for these codes is not necessary. Similar considerations are valid for the different full-potential transonic codes. There a vectorization may be performed for the subroutines containing the algebraic systems solvers, for example a Successive Line Overrelaxation (SLOR) method (see section 6.3).

In recent years a strong focus has been on Euler and Navier-Stokes code development. Several methods are under continual refinement. For the Euler equations the most widely used methods fall into two broad classes:

1. Central-difference methods with dissipation terms added to enhance stability and to provide smoothing of shock profiles.
2. Upwind-differenced flux-splitting and total variation diminishing methods.

The first class of methods usually requires some undesirable tuning to obtain near-optimum values of the coefficients for the added dissipation terms. The methods are relatively simple compared to those of the second class, which require less tuning. Both classes are extendible to the viscous Navier-Stokes equations. Vectorization for the explicit methods such as Runge-Kutta time-stepping Finite-Volume algorithms [52] is straightforward. For the implicit methods based on e.g. Beam and Warming's scheme (ARC3D, [97]) vectorization is performed in planes normal to the surface of the body. An example is given in section 6.6.

Parabolized Navier-Stokes (PNS) codes are being used extensively to compute steady supersonic and hypersonic flow about streamlined bodies. In the parabolizing approximation the streamwise viscous terms are dropped and a modified streamwise pressure gradient is introduced to allow space marching from upstream initial data. Because only one pass through the grid is required these procedures are computationally efficient. Current development activities with PNS are centered about the inclusion of real gas effects and finite-rate chemistry. Vectorization again may be carried out in planes normal to the surface, where often systems of equations could be solved in parallel. The cost of real Navier-Stokes applications has restricted their use to specialized applications, primarily 2-D flows and limited regions of 3-D flows. Most of the applications so far have used simple algebraic turbulence models with the Reynolds-averaged, thin-layer Navier-Stokes (TLNS) approximation. The TLNS uses body-fitted meshes so that all of the diffusion terms tangential to the solid boundaries can be conveniently dropped from the equations. This eliminates the need to treat meshes that are fine enough in all directions to resolve properly all the viscous derivatives. A crude estimate of the number of mesh points required to obtain a reasonably accurate solution of the TLNS is twice the number of mesh points as for a good Euler solution. Computer time required for present TLNS codes for even an isolated wing ranges from 2 to 8 hours on a CYBER 205 or CRAY X-MP. Because of the importance of TLNS codes for today's CFD we give a more detailed description on TLNS simulations using vector computers in the next section.

Algorithm development for full 3-D Navier-Stokes equations is still in a primitive state. Algorithms need to be improved in speed of convergence (from thousands of iterations down to hundreds) without suffering serious loss in performance and robustness to high aspect ratio meshes and artificial viscosity. New efficient, solution-adaptive meshes should track and resolve shocks, vortex structures, boundary layers, wakes, and free shear layers. Work to date has been mostly with global, nonadaptive, body-fitted grids for single components such as an airfoil, wing, isolated nacelle or hemisphere cylinder. However, promising research has shown the possibilities afforded by overlapping embedded grids, zonal body-fitted grids, and finite-element-type tetrahedral grids.

For unsteady aerodynamic flows - structural dynamics, flutter, and active controls - less emphasis has been on the development of CFD methods. Several of the methods in common use for steady aerodynamics can be used in a time-accurate, unsteady mode. To date computing costs required to carry out time-accurate calculations, however, are prohibitive except for perhaps very specialized applications.

CODES FOR HYPERSONIC APPLICATIONS

A major complication of hypersonic CFD in the upper atmosphere is that the continuum fluid model is not realistic for all hypersonic flight conditions of interest. At higher altitudes where the mean free path of the molecules becomes of sufficient magnitude relative to the vehicle dimensions, the continuum model breaks down. Here particulate flow simulations must be employed wherein the motion of a large number of molecules is computed, such as in the direct Monte Carlo simulation. Vectorization of this type of methods is very cumbersome, and up to now not very efficient. An example for the vectorization of a Monte Carlo code calculating the flow around a flat plate is given in [40].

Another complication for hypersonic flight is that the conventional continuum Navier-Stokes equations even in the lower atmosphere can be unrealistic or are of uncertain accuracy .

Hypersonic aircraft configurations are geometrically less complicated, and are thus more amenable to realistic 3-D grid generation. For flight in the lower atmosphere with equilibrium air chemistry, the feasible CFD applications would be generally similar to those mentioned previously for conventional aircraft.

In this case only a modest extension of supersonic aircraft CFD is necessary wherein real gas, equilibrium, thermodynamic properties of air are used in place of a perfect gas. However, for complex 3-D flows involving separated flow or inlet and airframe integration, for example, current codes are far from maturity. Codes for hypersonic flight in the upper atmosphere require much more computation time because of the numerous species continuity equations that must be solved simultaneously with the equations of fluid motion. Thus, the efficiency of numerical algorithms is an important aspect of this type of code development, and optimal vectorization for these codes is necessary but not straightforward.

CODES FOR PROPULSION APPLICATIONS

Propulsion system elements can be categorized into three major systems, namely stationary systems including inlets, nacelles, diffusers, and nozzles; rotational systems including fans, compressors, turbines, propellers, and helicopter rotors; and finally combustors including both combusting and noncombusting elements. Because this complexity of propulsion systems is difficult to reproduce experimentally advanced computer programs are used both to design an experiment and to predict the results, so that CFD plays the added role of a basic data source, helping to identify the dominant physics. For propulsion applications a wide variety of CFD codes are available based on different models such as inviscid models (full-potential and Euler equations), classical boundary-layer approaches (transitional and turbulent), inviscid interaction approaches, using both integral and finite difference (or volume) techniques, and composite viscous approaches employing the parabolized Navier-Stokes equations with spatial marching procedures for both subsonic and supersonic flows. Vectorization of these codes is performed planewise, very similar to those mentioned in the previous section for the thin-layer Navier-Stokes approximation.

The general approach used to attack the problems in combustors employs algorithms to solve the full turbulence-averaged Navier-Stokes equations. Here the two principal difficulties appear to be elimination of numerical diffusion and modeling of real turbulent diffusion. A better numerical representation of 3-D thin viscous shear layers between regions of forward and reverse rotational flows is needed to avoid the need for massive computer resources in representing the critical physics. To date, however, the use of the turbulence-averaged Navier-Stokes equations is generally considered too expensive for practical applications, even with the advent of the modern supercomputers.

TURBULENCE

Turbulence modeling is becoming a pacing item in the development of CFD codes for many engineering design applications. The large-scale features of turbulent flows are strongly dependent on the flow geometry and environment. Small-scale turbulence, which adjusts more rapidly to changes, tends to be more isotropic but also reflects the environment in which it is embedded. The large-scale motions are responsible for turbulent transport, and the small-scale motions provide intimate molecular mixing. With a turbulence MODEL one tries to represent the average effects of these complex processes on the mean flow. In turbulence SIMULATION one includes the large-scale, time-dependent motions in the computation.

The unsteady Navier-Stokes equations are assumed to describe the flow in full detail, including viscous flow phenomena and diffusive energy and species transport. Direct numerical solution of these equations for practical engineering flows at moderate and high Reynolds numbers, accurate to all temporal and spatial scales of motion, to date is impossible. Most engineering computations, instead, are based on the averaging procedure developed by Reynolds, which is called turbulence-(or Reynolds-) averaged Navier-Stokes. Various other averaging methods are in use or are expected to be used in the near future. A complete discussion of all categories of turbulence models (such as algebraic and K-epsilon models) will be omitted here. In many of today's codes the significance of turbulence modeling is masked by excessive numerical diffusion in the algorithms. Once this numerical diffusion is eliminated the significance of the turbulence model will become clearly evident. It is anticipated that turbulence modeling will be an essential element in CFD codes for the near future. Unfortunately, methods of improving turbulence modeling (e.g. full turbulence simulations or large eddy simulations) are extremely expensive because of very heavy computations on even the fastest available supercomputers and because of the sophisticated procedure of implementation.

CFD AND COMPUTER HARDWARE

Taking into account the evolution of CFD and computer hardware during the last 15 years, certain future trends are evident. As the cost of fast computer memory decreases, the appearance of memory-intensive models using many millions of words of primary data is already beginning to influence algorithm selection in favor of faster, more efficient algorithms that make less use of each computational degree of freedom, which means that much more data will have to be analyzed and stored in the future. Fine-grained parallelism, through pipelining and vector processing, has taken over the supercomputing scene in the last 15 years. Because of the limitations imposed by the speed of light and component technology, the trend toward increasingly extensive parallelism will continue in the future. Several supercomputers are multiprocessors, and already systems with thousands of small processors are being produced and sold commercially. Different algorithms will appear to be most favorable for this kind of supercomputing, and new programming methodologies will have to be adopted. In some cases, simpler, less-accurate solution algorithms and grids will be adopted because they lend themselves better to computation on highly parallel processors. In other cases the mathematical models themselves will evolve to reflect the computer systems on which they are being implemented.

Furthermore, interactive graphics methods and the associated hardware and software will become absolutely crucial to monitoring, interpreting, understanding, and presenting the computational results. Although these are not new issues, they are becoming dominant considerations for the future.

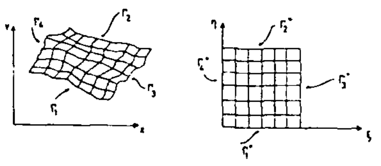
6.3 VECTORIZATION OF A MESH-GENERATION CODE

For the numerical solution of systems of partial differential equations, like the Navier-Stokes system, the computational domain is discretized into many subdomains. On an uniform grid the equations are easily discretized. However, in the case of a body with arbitrary shape, the numerical treatment of the boundary conditions is greatly simplified only by the use of a curvilinear or even irregular computational mesh system, which is conformable to the body surface, as well as to other boundaries, in such a way that the boundaries are mesh lines [20],[123].

Among the many possible choices for the partial differential equations to be solved in the grid generation process most investigations to date have used sets of elliptic equations derived from Laplace's or Poisson's equation [123]. The use of elliptic equations, particularly those whose solutions only have extrema on boundaries, such as Laplace's equation, has several attractive features. Firstly, boundary data must be specified on all boundaries, thus this method is particularly suited to constrained (e.g. internal) flows or flows where an outer free stream boundary can be specified, while the extremum principle ensures that the mapping is univalent (i.e. grid lines do not cross over). Furthermore, since Laplace's equation describes a range of physical phenomena, physical considerations may sometimes help in choosing suitable grid control parameters. For example, with suitable boundary conditions the grid obtained consists of streamlines and equipotential lines and the addition of extra source or sink terms (i.e. replacing Laplace's equation by Poisson's equation) allows some control over the shape of these lines. Finally, it should be noted that conformal mapping in two dimensions is a special case of grid generation using Laplace's equation.

Although some work had been reported previously, the major investigation and development of grid generation using Laplace's and Poisson's equations has been done by Thompson (1979) and his co-workers [123].

Figure 6.1: Basic mapping using elliptic equations [20].



To describe the basic Thompson technique using Poisson's equation consider the case shown in Fig.6.4 in which a general quadrilateral in physical space is mapped to a unit square in transform space. The coordinates in physical space are (x,y) and in transform space are (u,v) and it is assumed that x and y are known functions of u and v on all the boundaries (i.e. the grid point distributions are specified along the boundaries). Assume that u and v individually satisfy Poisson's equation (expressed in terms of x and y) inside the unit square subject to the known Dirichlet conditions on the boundaries, i.e.

$$\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = P$$

$$\partial^2 v/\partial x^2 + \partial^2 v/\partial y^2 = Q$$

where the source terms P and Q are both functions of x and y. In order to obtain a grid in physical space, it is necessary to interchange the dependent and independent variables so that the equations may be solved numerically for x and y on an equally spaced grid in transform space. The Dirichlet boundary conditions for the inverted equations are the known values of x and y at each point on the boundary of the unit square in transform space. Note that in tensor form the extension to three dimensions is immediate.

The inverted equations can, in principle, be solved by any convenient method. However, in practice the usual approach has been to discretize the equations using three point central difference approximations for all derivatives and solve the resulting set of difference equations using a line relaxation scheme.

Such schemes are widely used in scientific and engineering codes. Because of their highly recursive nature, we present restructuring with respect to vector architecture in more detail. The most general approach for block iterative methods resulting from systems of partial differential equations (sometimes called Richtmyer's algorithm) will be presented in chapter 6.6 in the context of Beam and Warmings approximate factorization scheme.

We now start from the transformed elliptic equations

$$A\partial^2 x/\partial u^2 + B\partial^2 x/\partial v^2 - C\partial^2 x/\partial u\partial v + \partial P\partial x/\partial u + Q\partial x/\partial v = 0$$

$$A\partial^2 y/\partial u^2 + B\partial^2 y/\partial v^2 - C\partial^2 y/\partial u\partial v + P\partial y/\partial u + Q\partial y/\partial v = 0$$

with metric coefficients A, B and C containing themselves derivatives of the solution x and y. These equations can be discretized using central difference quotients (with delta u = delta v = 1 in the computational space). The result is (for the unknown x) the following difference equation

```
A * (X(I+1,J) - 2 * X(I,J) + X(I-1,J)) +
B * (X(I,J+1) - 2 * X(I,J) + X(I,J-1)) -
0.25 * C * (X(I+1,J+1) - X(I+1,J-1) - X(I-1,J+1) + X(I-1,J-1))
+ 0.5*P*(X(I+1,J) - X(I-1,J)) + 0.5*Q*(X(I,J+1) - X(I,J-1)) =0
```

and analogously for the unknown y. Usually a relaxation scheme (with relaxation factor OM) is applied to these equations:

$$
\begin{aligned}
X^{(n+1)}(I,J) = X^{(n)}(I,J) &+ OM1*(A*(X^{(n+1)}(I+1,J) - 2*X^{(n+1)}(I,J) + X^{(n+1)}(I-1,J)) \\
&+ B*(X^{(n)}(I,J+1) - 2*X^{(n)}(I,J) + X^{(n+1)}(I,J-1)) \\
&- 0.25*C*(X^{(n)}(I+1,J+1) - X^{(n+1)}(I+1,J-1) + X^{(n)}(I-1,J+1) + X^{(n+1)}(I-1,J-1)) \\
&+ 0.5*P*(X^{(n)}(I+1,J)-X^{(n)}(I-1,J))+0.5*Q*(X^{(n)}(I,J+1)-X^{(n)}(I,J-1))))
\end{aligned}
$$

for every line J = const. and *OM1 = OM/(2*B). This is a system of algebraic equations* for the unknown X(I,J) , I = 2,3,..., N-1. It is linear if the coefficients A,B, and C are evaluated at the previous iteration step n. Solving these equations for the differences

$$RX(I,J) = X^{n+1}(I,J) - X^{n}(I,J)$$

simplifies the systems to

$$
\begin{aligned}
&-A*RX(I-1,J) + (2*A+2*B/OM) * RX(I,J) - A*RX(I+1,J) \\
&= RES(I,J) + B*RX(I,J-1) + 0.25*C*(RX(I+1,J-1) - RX(I-1,J-1))
\end{aligned}
$$

for every J = 2,3,..., N-1. RES(I,J) is the (residual) value of the difference equation at the grid point (I,J). These systems for the RX(I,J) then are solved by the Thomas algorithm (see chapter 5). The final solution at the iteration level n+1 is

$$X^{n+1}(I,J) = X^{n}(I,J) + RX(I,J).$$

As pointed out in chapter 5, the Thomas algorithm is *not suited for vector computers* because of its highly recursive structure. A slightly modified version for the matrix

$$
A = \begin{bmatrix}
dd1 & aa1 & & & \\
aa2 & dd2 & aa2 & & \\
 & \cdot & \cdot & \cdot & \\
 & & \cdot & \cdot & \cdot \\
 & & & aam & ddm
\end{bmatrix}
$$

with only N divisions (instead of 2N-1 for the original algorithm) may have the following form:

```
   D(1)    = 1. / DD(1)
   DO      40     I = 2,N
   R(I)    = AA(I) * D(I-1)
   D(I)    = 1. / (DD(I) - AA(I-1) * R(I))
   RX(I)   = RX(I) - RX(I-1) * R(I)
40 CONTINUE
   RX(N)   = RX(N) * D(N)

   DO      50   K = M-1, 1, -1
   RX(K)   = (RX(K) - AA(K) * RX(K+1)) * D(K)
50 CONTINUE
```

with, at the beginning, RX containing the right-hand side of the system. Vectorization of the three recurrences may be achieved, for example, by solving all the systems for J = 2,3,..., N-1 in parallel. However, running over the second array index J will cause data to be fetched from memory with a large stride, which has been observed to degrade performance in many vector architectures. It is, therefore, more appropriate to solve systems of equations along lines I = const. in parallel then running over the first array index I = 2,3,..., N-1. The corresponding relaxation scheme for lines I = const. yields (with OM1=OM/(2*A))

```
         (n+1)  (n)             (n)        (n)     (n)
        X(I,J)=X(I,J)+OM1*(A*(X(I+1,J)-2*X(I,J)+X(I-1,J))

                                  (n+1)       (n+1)  (n+1)
                              +B*(X(I,J+1)-2*X(I,J)+X(I,J-1))

            (n)         (n)         (n)         (n)
  -0.25*C*(X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1))

           (n)        (n)                 (n)       (n)
  +0.5*P*(X(I+1,J)-X(I-1,J))+0.5*Q*(X(I,J+1)-X(I,J-1)))
```

Comparing this with the original relaxation scheme one recognizes that the first scheme will converge faster because more unknown values are updated immediately (6 compared to 3 for the second scheme). The reason is that the neighboring values for I-1 and I+1 are not yet available for the iteration level n+1 because of the parallel evaluation of the systems. Therefore, convergence for the latter algorithm will be slower which is more than compensated by the much faster execution speed on most vector computers. Introduction of the differences

```
                (n+1)    (n)
    RX(I,J) = X(I,J) - X(I,J)
```

leads to the systems

```
  -B*RX(I,J-1)+(2*B+2*A/OM)*RX(I,J)-B*RX(I,J+1)=RES(I,J)
```

for every I = 2,3,...,N-1. The final solution at the iteration level n+1 is

```
   (n+1)     (n)
  X(I,J) = X(I,J) + RX(I,J).
```

As mentioned earlier, these systems can be solved in parallel using vectors with element indices I = 2,3,...,N-1, pointing to contiguously stored elements. The parallel Thomas algorithm then reads

```
      DO      400      I = 2,N-1
400   D(I,1)   =   1./ DD(I,1)

      DO      401      J = 2,N
      DO      401      I = 2,N-1
      R         = AA(I,J) * D(I,J-1)
      D(I,J)   =1./( DD(I,J) - AA(I,J-1) * R)
401   RX(I,J) = RX(I,J) - RX(I,J-1) * R

      DO      411      I = 2,N-1
411   RX(I,N) = RX(I,N) * D(I,N)

      DO      501      K = M-1,1, -1
      DO      501      I = 2, M-1
501   RX(I,K) = (RX(I,K)- AA(I,K) * RX(I,K+1)) * D(I,K)
```

This system is fully vectorized and runs with maximum execution speed. However, as pointed out before, the convergence is worse compared to the original relaxation scheme because of the parallel evaluation of all systems I = 2,3,..., N-1. This can be improved by introducing a ZEBRA structure within the algorithm, thus solving, in a first half iteration step, the systems for all even I = 2,4,..., N-1 and in a second half iteration step, the systems for all odd I = 3,5,..., N-2. For the even systems, the odd neighbors have been updated just before, and vice versa. This, then, leads to a much better convergence behavior of the algorithm. On the other hand, vector performance is somewhat reduced because data now are fetched with a stride of 2 and the vector length is halved. The overall improvement, therefore, depends very much on the special vector computer architecture.

6.4 THIN LAYER NAVIER STOKES SIMULATIONS ON VECTOR COMPUTERS (Dr. J. L. Steger)

Over the last several years, versatile computer codes have been developed for large scale computers which can solve steady or unsteady and inviscid or viscous flow c.f. [131]-[144]. These codes are based on using finite difference and finite volume approximations to the Euler and Navier-Stokes equations, and they generally use curvilinear body conforming discretization processes. These methods have proven to be quite satisfactory insofar that wave propagation, viscous layers, shock waves, and unsteady motions can all be treated. In high Reynolds number viscous flow simulation, the use of a body conforming curvilinear discretization process is essential to simplify the application of boundary conditions, to efficiently cluster to thin shear layers near the wall, and to make simplifying assumptions such as the thin layer viscous flow approximation. Generally a structured (well-ordered) grid is used for the curvilinear discretization and such a well ordered grid enhances the use of vector computers. It also allows use of certain efficient numerical solution schemes that rely on directional splitting techniques such as locally one dimensional split schemes, alternating direction implicit (ADI) and approximate factorization (AF).

6.4.1 Governing Equations

On a body conforming curvilinear coordinate the Euler and Navier-Stokes equations can be solved by the integral or finite volume method, or the finite difference method by introducing new independent variables. In either case the equations are solved in a conservation or divergence form to allow the capturing of shock waves as numerically accurate as possible.

In the new independent variables the transformed equations can be represented as (c.f. [133], [145] for the detailed terms):

$$\partial_\tau \hat{Q} + \partial_\xi(\hat{F} + \hat{F}_v) + \partial_\eta(\hat{G} + \hat{G}_v) + \partial_\zeta(\hat{H} + \hat{H}_v) = 0 \qquad (1)$$

where the original dependent velocity variables are maintained. (The flux vectors of the transformed equations can be made to resemble their Cartesian counterparts by combining terms into contravariant velocity components [133].) If a body conforming coordinate is used, then for high Reynolds number flow it is generally permissible to make a thin layer assumption and to discard viscous terms except for those in the normal-like direction. If ζ is the coordinate away from the surface, the thin layer equations can be represented as [133]

$$\partial_\tau \hat{Q} + \partial_\xi \hat{F} + \partial_\eta \hat{G} + \partial_\zeta \hat{H} = Re^{-1}\partial_\zeta \hat{S} \qquad (2)$$

where the viscous terms in ζ have been collected into the vector $\hat{S}$ and the nondimensional reciprocal Reynolds number is extracted to indicate a viscous flux term.

In differencing these equations it is often advantageous to difference about a base solution denoted by subscript $_0$ as

$$\begin{aligned}\delta_\tau(\hat{Q} - \hat{Q}_0) + \delta_\xi(\hat{E} - \hat{E}_0) + \delta_\eta(\hat{F} - \hat{F}_0) + \delta_\zeta(\hat{G} - \hat{G}_0) - Re^{-1}\delta_\zeta(\hat{S} - \hat{S}_0) \\ = -\partial_\tau \hat{Q}_0 - \partial_\xi \hat{E}_0 - \partial_\eta \hat{F}_0 - \partial_\zeta \hat{G}_0 + Re^{-1}\partial_\zeta \hat{S}_0\end{aligned} \qquad (3)$$

where δ indicates a general difference operator, and ∂ is the differential operator. If the base state is properly chosen, the differenced quantities can have smaller and smoother variation and therefore less differencing error.

6.4.2 Implicit Finite Difference Algorithms

Both explicit and implicit numerical algorithms have been written to solve these equations. Generally implicit solution algorithms are preferred for viscous flow to avoid excessively restrictive numerical time step restrictions in unsteady flow calculations and to enhance iterative convergence in steady state calculations. (Various ways to enhance essentially explicit schemes include use of multigrid, e.g. [146] and [147] and various embedding schemes, e.g. [148]. Even for inviscid flow problems in which the grid spacing is not as refined, the use of implicit schemes can be advantageous insofar that time step restrictions can be based on accuracy considerations alone. Moreover one need not be too concerned if the grid spacing is inadvertently too refined or distorted.

Because the Euler or Navier-Stokes equations are nonlinear, a fully implicit scheme either requires that for each time step taken an iterative solution process be carried out, or a local linearization be carried out based on a previous time step and that a complete inversion of the linearized equations be carried out. Both procedures are costly and generally some type of simplification is used that falls within the accuracy of the differencing process. Typical of these is the approximate factorization method in which the local linearization matrix is approximately factored to within the order of time differencing accuracy in such a way that the inversion work per step is greatly reduced. Two such schemes for Eq.(3) are illustrated below.

Equations (2) or (3) have been solved using a Beam-Warming noniterative approximate factorization implicit scheme of the form, c.f. [133], [149]-154],

$$\left[I + h\delta_\xi \hat{A}^n - D_i|_\xi\right]\left[I + h\delta_\eta \hat{B}^n - D_i|_\eta\right]\left[I + h\delta_\zeta \hat{C}^n - hRe^{-1}\bar{\delta}_\zeta J^{-1}\hat{M}^n J - D_i|_\zeta\right]\Delta\hat{Q}^n$$
$$= -\Delta t\left[\delta_\xi(\hat{E}^n - \hat{E}_\infty) + \delta_\eta(\hat{F}^n - \hat{F}_\infty) + \delta_\zeta(\hat{G}^n - \hat{G}_\infty) - Re^{-1}\bar{\delta}_\zeta(\hat{S}^n - \hat{S}_\infty)\right]$$
$$- D_e(\hat{Q}^n - \hat{Q}_\infty)$$

where h = Δt or (Δ t)/2 and the free stream base solution is indicated. Here δ is typically a three point second order accurate central difference operator, while $\bar{\delta}$ is a midpoint operator used with the viscous terms. The matrices $\hat{A}$, $\hat{B}$, $\hat{C}$, and $\hat{M}$ result from local linearization about the previous time level and J is the Jacobian of the coordinate transformation. The factored left hand side operators form block tridiagonal matrices. Because central space difference operators are used, numerical dissipation terms denoted as D_i and D_e have been inserted into Eq.4 In their simplest form these have been given as combinations of second and fourth differences, c.f.[151].

As an alternative to using central space differencing for the convection terms, upwind (i.e. backward and forward) space differencing can be used (c.f. [134], [137], [152]-[126]) if the fluxes are properly split according to their characteristic properties. Upwind schemes can have several advantages over central difference schemes, including natural numerical dissipation, better explicit stability, and more readily inverted implicit schemes. Conversely, upwind schemes for systems of equations have generally been more complicated and computationally expensive than central difference schemes and are not very suitable for treating viscous terms.

A mixed upwind-central difference scheme can be very effective if the upwind differencing is used in the streamline direction. An implicit algorithm for the thin layer Navier-Stokes equations using flux-splitting and upwind differencing in one direction is given by [156]

$$\left[I + h\delta_\xi^b(\hat{A}^+)^n + h\delta_\zeta \hat{C}^n - hRe^{-1}\bar{\delta}_\zeta J^{-1}\hat{M}^n J - D_i|_\zeta\right]$$
$$\times\left[I + h\delta_\xi^f(\hat{A}^-)^n + h\delta_\eta \hat{B}^n - D_i|_\eta\right]\Delta\hat{Q}^n =$$
$$-\Delta t\{\delta_\xi^b[(\hat{E}^+)^n - \hat{E}_\infty^+] + \delta_\xi^f[(\hat{E}^-)^n - \hat{E}_\infty^-] + \delta_\eta(\hat{F}^n - \hat{F}_\infty)$$
$$+\delta_\zeta(\hat{G}^n - \hat{G}_\infty) - Re^{-1}\bar{\delta}_\zeta(\hat{S}^n - \hat{S}_\infty)\} - D_e(\hat{Q}^n - \hat{Q}_\infty) \quad (5)$$

Here δ_ξ^b and δ_ξ^b are backward and forward three-point difference operators and D_e contains only η and ζ numerical dissipation operators. This two-factor implicit scheme is readily vectorized or multi-tasked in planes of ξ = constant. A semi-implicit scheme is obtained by neglecting the calculation of h $\delta_\xi^f \hat{A}^-$ in the implicit backsweep operating on ΔQ^n.

Body surface boundary conditions have often been supplied to the above algorithms by using a combination of normal-momentum, tangency or no slip, and extrapolation [133]. Various farfield conditions have been used including characteristic-like conditions. Because of their simplicity, in most of the application codes the boundary conditions have been imposed explicitly, or a combination of simplified implicit-explicit conditions have been used. However, fully implicit boundary conditions have also been used ans an elegant implicit characteristic-like formulation has been described by Chakravarthy [157].

6.4.3 Vectorization and Multi-Tasking

The structure of the above algorithms lends itself to vectorized computer coding. All of the operations to the right hand side of the equal sign simply require that difference operations be carried out using known data on a well-ordered grid. Like an explicit algorithm, these operations are very simple to vectorize. To the left hand side of the equal sign, implicit operations must be carried out, however, because of the approximate factorization, only solution of uncoupled sets of block tridiagonals (with 5 x 5 blocks) have to be carried out. Writing equation (4) in an algorithm form helps to clarify

$$\left[I + h\delta_\xi \hat{A}^n - D_i|_\xi\right] \Delta\hat{Q}^{**} = RHS$$

$$\left[I + h\delta_\eta \hat{B}^n - D_i|_\eta\right] \Delta\hat{Q}^{*} = \Delta\hat{Q}^{**}$$

$$\left[I + h\delta_\zeta \hat{C}^n - hRe^{-1}\bar{\delta}_\zeta J^{-1}\hat{M}^n J - D_i|_\zeta\right] \Delta\hat{Q}^n = \Delta\hat{Q}^{*}$$

$$\hat{Q}^{n+1} = \hat{Q}^n + \Delta\hat{Q}^n$$

Interpretation of this algorithm shows that in the first line, the RHS must be formed over the field. The values can be temporarily stored in the array that will hold $\Delta\hat{Q}$. The left hand side operator of the first line forms a block tridiagonal matrix (with 5 x 5 blocks) in ξ for each $\eta - \zeta$ grid line intersection. For example, if indices j,k,l correspond to ξ, η, ζ the first factor of the equation forms a block tridiagonal between, say, points j = 1 to j = jmax. There is one such ξ-block tridiagonal for each k,l index. Efficient solution routines for block tridiagonal matrices are inverted simultaneously so as to avoid the recursive nature of matrix elimination procedures. Because there is a ξ-block tridiagonal for each k,l index, vectorization can be achieved by simultaneously inverting ξ-block tridiagonals over, say, k = 1,kmax. In this way a vector length of kmax is achieved, see Fig. 6.2 .

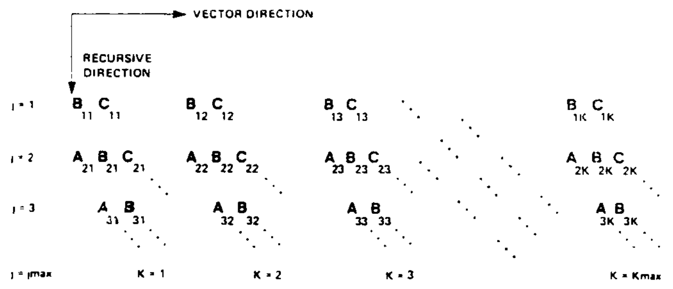


Figure 6.2: Simultaneous inversion of block tridiagonals to obtain a vector length.

However, in inverting a ξ-block tridiagonal, temporary storage is needed for the backward elimination; and this storage requirement is increased by kmax. (Means of reducing the block size are discussed in [158] and [159] as a way to improve efficiency, and these same techniques reduce temporary storage as well). Overall, vectorization tends to complicate the coding, but not unduly so. On the CRAY-XMP, vectorized codes for the above equation runs about 5 times faster than the unvectorized code for a typical application, see [151]. Additional details on the vectorization of this scheme are given in section 6.6 and in [160].

The same vectorization approach works for the mixed upwind scheme given by equation 5. For the first factor, efficient inversion requires recursive operations in both the ξ and ζ directions. This is because the ξ-operator is a backward differencing and forms a lower triangular matrix which is readily solved by sweeping through the grid from j = 1 to jmax while the ζ-operator forms a block tridiagonal as well. However, η-operators do not appear in the first factor, so vectorization is achieved at a given ξ-plane by simultaneously inverting in η the block tridiagonals formed in ζ. The second factor is treated in the same way. Now a backsweep through the grid is required and block tridiagonals formed in η are inverted simultaneously in ζ .

The above implicit algorithms can also be coded for multi-tasking by assigning each processor a portion of the code. Each processor need only be assigned some segment of the uncoupled block tridiagonals. However, because of the extra recursiveness in the mixed upwind-central scheme, only one direction can be partitioned without creating explicitly lagged boundaries. In this case the vector length that each processor deals with becomes smaller as more processors are used.

Improvements to the basic algorithm in both efficiency and accuracy have been made by a variety of contributors. To improve its overall efficiency, simple changes have been optionally implemented. They include the use of space varying Δt, use of a sequence of coarsened grids to provide a good initial guess, cutting inversion costs by using either diagonalization [158] or block reduction [159] methods, implementation of better numerical dissipation terms, and more implicit treatment of the numerical dissipation terms. As described in [151], these combined changes can improve steady state efficiency by an order of magnitude.

Although the implicit algorithm has been presented with three point central differencing, versions of the algorithm that have fourth order accuracy in space have been available [133] and are preferred unless strong shock waves are present. Reddy [131] has also demonstrated a version in which a pseudo-spectral operator is used in place of the right hand side convection operators. Total variation diminishing (TVD) implementations [132] and [133] have also been carried out to better capture shocks, and perturbation about approximate base solutions has been used to reduce the number of needed grid points{[134] and [135]}.

6.4.4 Composite Mesh Schemes

The previous algorithms have been described for use on a single body conforming curvilinear grid. This means that a structured body conforming, smoothly varying, and properly clustered grid must be generated. Or restated another way, the generated grid must not be discontinuous, too skewed, and it should not waste points in regions of the field in which little change takes place. For simple configurations such a grid is relatively easy to generate. However, it has not been feasible to generate a single body conforming grid that is practical in the way described above for a complex three dimensional configuration such as an airplane. Using a single grid, grid lines simply become too skewed or too poorly clustered.

One way to extend the algorithms to complex configurations is by using composite grids. Composite grids use more than one grid to mesh an overall configuration with each individual grid of the system patched together or overset. The sketches shown in Fig. 6.3 illustrate several simple patched and overset grid configurations in two dimensions for a typical two body problem. As the sketches illustrate, patched grids are individual meshes that are joined together at some common interface surface. With overset grids the meshes are simply superimposed or partially superimposed to cover the region of interest, and are not joined together in any special way, although they can be.

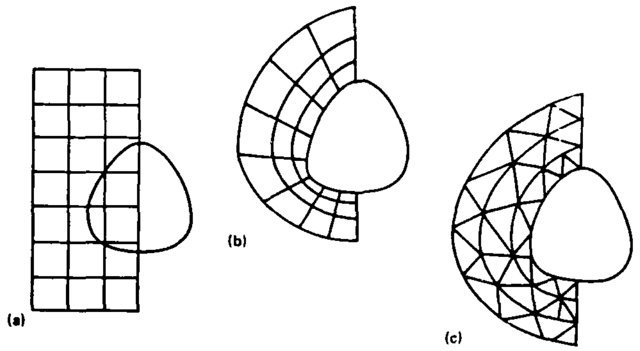


Figure 6.3: Sketches showing basic grid treatments for a simple body : a. rectangular or Cartesian; b. body conforming curvilinear; c. body conforming irregular triangularized.

The use of a set of patched or overset grids to form a larger composite grid carries the discretization process one step further. In a sense a composite mesh scheme assumes some of the characteristics of an unstructured grid in which the overall grid is made up of a few well-ordered grids that are tied to each other in an unstructured way. Because each individual grid in the system is well-ordered, each is suitable for efficient finite difference solution using vectorized computers and any available single grid code. The problem with a composite grid scheme is the difficulty of accounting for all the possible communications between meshes and the difficulty of supplying interface boundary data without degrading numerical accuracy or convergence.

Limited experience with both patched and overset grids (e.g., [136], [139], [141]) has not shown which method is preferable. An optimum method will likely combine both patched and overset grids and perhaps small grid segments that are not well ordered, and such grids have already been tried [166]. Both patched and overset grid schemes necessitate extensive bookkeeping procedures. One of the drawbacks of the patched grid method is a grid generation problem that is still relatively difficult because various interfaces have to be defined and grids with both inner and outer boundary surfaces must be generated. Drawbacks to overset grids include interpolation of data points along an irregular boundary and bookkeeping which can be especially complex if more than two levels of overset grids intersect each other.

With composite grids the possibility exists of using different computers to update the results on each grid if a multitasking computer processor is available, but this is feasible only if the amount of work on each grid or subdivided grid is roughly the same. Likewise, on machines which have a small but high speed memory and a large but low speed peripheral memory it is feasible to keep only the memory requirements for a given grid internal to the small fast memory. Of course, one can partition a single grid scheme for multitasking and memory can be rolled back and forth, but with composite grid schemes this data management is naturally built into the code.

6.5 VECTORIZATION OF MACCORMACK'S METHODS

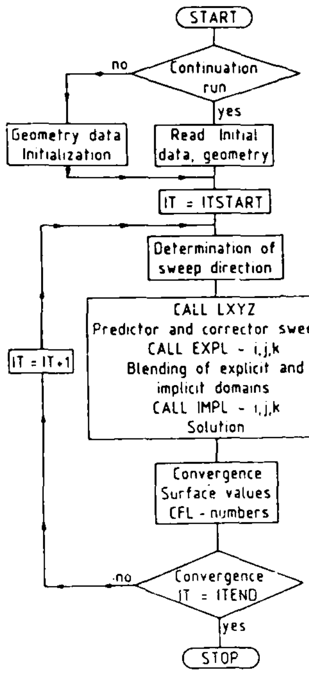
MacCormack's purely explicit-corrector versions [77] of the two-level scheme of Lax and Wendroff are easily applied, and can, in general, be completely vectorized [25,112], but they have to satisfy severe stability conditions with respect to the marching step size. For the explicit method to be stable in time-dependent calculations, the time-wise step must be chosen proportional to the square of the corresponding spatial step size divided by the kinematic viscosity. In the case of inviscid flow described by the Euler equations, the time step size has to be proportional only to the spatial step size itself divided by the appropriate velocity, this being the Courant-Friedrichs-Lewy condition.

In the mid-seventies, the development of fully implicit schemes was favored for integrating the time-dependent, governing equations for aerodynamic flows, because, according to the stability analysis of linear model equations, such methods are not restricted to small time steps. In practice restrictions do occur due to the method of implementing the boundary conditions, due to the non-linearity of the equations, or, of course, just due to the necessity of resolving physical features of the flow in question.

MacCormack's explicit-implicit scheme [61,79] is based on the original, explicit method [77]. A predictor-corrector, implicit operator is incorporated into the scheme to provide the capability of taking larger steps than are allowed by the explicit stability condition. It is sufficient to consider essentially only the Euler terms of the governing equations in deriving the implicit operator using some corrections for the neglected viscous terms. Thereby the computational effort is reduced considerably compared with time-accurate implicit schemes. The computational effort is further reduced by the fact that the implicit sequences are only applied where it is really necessary, and are otherwise skipped completely.

Both, explicit and implicit, predictor and corrector sweeps, are performed each in a separate subroutine: EXPL-n and IMPL-n for each direction Xn, n = 1,2,3. Each of the six subroutines consists essentially of three nested DO-loops, namely for the X1-, X2- and X3-direction, with the n-loop as innermost loop. Within this inner loop the flux across one surface of cell n is determined for each value of n. In the explicit routines the loop has always the range from n = 1 to n = (nmax - 1), and uses the surface normals at X(n+1) = constant; the appropriate predictor or corrector fluxes are obtained by either leaving as is, or adding an increment of one to n (as appropriate) to fetch the corresponding dependent variables. Note that for each value of n an IF-statement is necessary to determine whether or not the implicit procedure must be followed. The fluxes can be considered as local flux boundary condition and are determined at the end of the nth step. (See flow chart).

Figure 6.4: Flow chart of the explicit-implicit MacCormack code.



The current efforts to completely vectorize the explicit portions of the explicit-implicit scheme use the auto-vectorization feature of the CRAY compiler. Hence it was necessary to decompose long DO-loops into smaller vectorizable loops, to remove IF- and CALL-statements from DO-loops, to unroll short inner DO-loops to enable larger vector loops, and to use the direction with the largest number of points as innermost DO-loop variable. The introduction of dummy arrays is necessary to store intermediate scalars into vector strings which allows exploiting the vectorizing capabilities of the CRAY-1S. On the other hand, small, locally used arrays, are redefined into longer vector strings (for more details see [40]).

Using this strategy, the recursive bi-diagonal implicit sweeps have also been almost completely vectorized. The only exceptions are the three IF-statement loops necessary to decide whether and where the scheme is implicit. While on serial computers, it is quite reasonable to detect single locations. If multiple implicit spots occur in one direction, strings of locations are formed by searching for the first and last "implicit point" in each direction considered.

Table 6.4: Computation times in seconds for the vectorized Euler and laminar Navier-Stokes solutions per mesh cell and iteration.

Mesh	$31\times20\times31$ (19220)			$42\times20\times31$ (26040)	
RDP	Method				
	Explicit	Implicit		Implicit	
		Less	More		
Navier-Stokes					
Full	5.9	8.5	9.3	8.5	10^{-5} s
CFL_i		0.23 (0)	0.56 (5168)	0.24 (0)	
CFL_j		1.24 (1207)	3.03 (2139)	1.28 (1139)	
CFL_k		164 (8349)	390 (9543)	167 (11520)	
Thin-layer	2.5	5.7	6	5.5	10^{-5} s
CFL_i		0.23 (0)	0.56 (5168)	0.24 (0)	
CFL_j		1.24 (1270)	3.03 (2139)	1.28 (1139)	
CFL_k		164 (8349)	390 (9543)	167 (11520)	
Euler	2.1	4.9	5.1	5	10^{-5} s
CFL_i		0.22 (0)	0.40 (1366)	0.47 (3735)	
CFL_j		1.22 (1903)	2.22 (1377)	2.54 (1288)	
CFL_k		1.56 (3842)	2.85 (5418)	3.29 (7600)	

$(\cdot\cdot\cdot\cdot)$: number of cells, including boundary cells.
CFL_n: *maximum* (inviscid) CFL-number in direction x^n.
RDP: CPU-time/(cells × iterations).

Vector loops are introduced in one of those directions which do not coincide with the recursive direction. For different meshes one has then to take care that a direction is used which produces longest vector strings and the smallest number of DO-loops to be initiated in order to reduce the start up time for the initialization of the vector loop.

The increase in efficiency is expressed in terms of the rate of data processing RDP (RDP = CPU time/(total number of grid points or cells times total number of iterations), [113]. For the original code calculating the three-dimensional flow field past an inclined blunt body [63] the ratio of computer times on the IBM 3081K and on the CRAY-1S using scalar performance only, is roughly 7.8, which increases to about 10 when autovectorization is permitted. In comparison with the hand-vectorized version, a ratio of about 31.4 is achieved for the computation of laminar flow. Note that this ratio is for a semi-implicit version for the 31 x 20 x 31 mesh, where in the axial (31) direction the integration was completely explicit, and in the circumferential (20) direction the scheme was only semi-implicit.

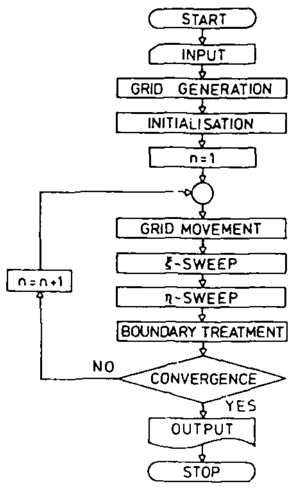
The table indicates that the increase of computational work due to the viscous effects of the thin-layer approximation is less than 20% of the work for the Euler equations. Note that the first step sizes away from the body in terms of radii are fairly small, namely 0.00005 for the viscous (laminar) and 0.005 for the inviscid case. The table shows to what extent the time requirements increase if the full Navier-Stokes solution is used instead of the thin-layer approximation. The increase is largest for the explicit calculations while the effect is less dramatic in the case of implicit integration. Note that the time requirements (per cell and time step) for the implicit solution remain nearly the same if the grid size is increased by roughly 30% [63].

6.6 VECTORIZATION OF THE IMPLICIT BEAM AND WARMING SCHEME

The implicit factored finite-difference scheme of Beam and Warming [9] is employed to solve the axisymmetric thin-layer Navier-Stokes equations. Approximating the time derivative by the first-order Euler implicit or the second-order three-point backward formula, linearizing the flux vectors by Taylor-series expansions, and using second-order, central differences to evaluate the metric terms and the spatial derivatives of the flow variables at interior grid points. leads to a finite-difference equation, which is approximately factored and implemented by a sequence of two one-dimensional matrix inversions referred to as the ξ-sweep and the η-sweep (cf. [9,97]).

All parts of this finite-difference method are easily vectorizable (section 6.4 and [40]) except the solution of the resulting block-tridiagonal linear systems. Applying the Richtmyer algorithm simultaneously instead of separately, also vectorizes the block-tridiagonal system solver, but at the cost of increased storage requirements. The computed example of axisymmetric laminar supersonic flow over a hemisphere-cylinder [85] demonstrates the advantages of vectorization. As high Reynolds number flows are to be examined, the thin-layer approximation of the Navier-Stokes equations is employed. The original version was written in FORTRAN IV and implemented on the IBM 3081K. From the flowtrace, the most time consuming subroutines were selected and modified following the general guidelines for vectorization on the CRAY-1S.

Figure 6.5: Flow chart for the solution of the axisymmetric thin-layer Navier-Stokes equations by the Beam and Warming scheme.



In the subroutines solving each block-tridiagonal linear system by the Richtmyer algorithm, the inversion of the diagonal matrices was originally performed in two subroutines decomposing and solving the corresponding linear systems, respectively. Pulling these subroutines, which were called nearly 2.7 million times in the example, in the calling routine, decreased the computing time for that subroutine by a factor of 17.36.

The computation of the block-tridiagonal matrices was not vectorized by the compiler because of the number and complexity of the 48 elements to be calculated in a single DO-loop. The remedy was to split the loop into three, and to calculate common temporary arrays in another DO-loop, thereby reducing the CPU-time of the corresponding subroutines by a factor of up to 2.89.

Unrolling inner loops of vector lengths 2,3, and 4 may decrease the CPU-time considerably. Unrolling small nested loops may even enhance chaining. Applying this guideline to the subroutine solving a block-tridiagonal linear system by the Richtmyer algorithm, resulted in a speed up factor of 12.59. As almost 80% of the total CPU-time of the original version of the blunt body code was spent in that subroutine, its modification reduced the overall computing time by factor of 3.8.

To compute the sum of certain residual vectors of all grid points in the subroutine checking the convergence, the CRAY-1S Fortran intrinsic function SSUM was employed. The use of SSUM and ISMAX, which determined the indices of the maximum residuals, led to a speed-up factor of 3.32 for this subroutine.

Because of the dependences due to the recursions in the Richtmyer algorithm, the subroutine solving a block-tridiagonal linear system reached a computing speed of only 14.45 MFLOPS on the CRAY-1S. The elimination of these dependences by modifying the Richtmyer algorithm, reduced the total CPU-time by a factor of 1.46 (see the following table).

Table 6.5: CRAY-1S CPU-times of blunt body code versions for 400 time levels on a 26x31 grid.

	Richtmyer version (s)	Simultaneous Richtmyer version (s)
Input	0.01	0.01
Grid generation	0.02	0.02
Initialization	0.03	0.03
Grid movement	0.40	0.40
ξ-sweep	11.88	7.59
RHS	3.73	3.56
Matrices	1.25	1.16
Richtmyer	6.83	2.82
Store	0.07	0.05
η-sweep	11.16	7.20
Matrices	4.20	4.11
Richtmyer	6.86	3.01
Store	0.10	0.08
Boundary treatment	0.29	0.29
Convergence	0.54	0.54
Output	1.91	1.91
Total	26.24	17.99

SOLUTION OF BLOCK-TRIDIAGONAL LINEAR SYSTEMS ON VECTOR COMPUTERS

(a) Richtmyer algorithm:

The solution of the block-tridiagonal linear system

(1) $A\ x = f$

is to be found, where

$$A = \begin{pmatrix} A_1 & C_1 & & & \\ B_2 & A_2 & C_2 & & \\ & B_3 & A_3 & C_3 & \\ & \ddots & \ddots & \ddots & \\ & & B_{K-1} & A_{K-1} & C_{K-1} \\ & & & B_K & A_K \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{K-1} \\ x_K \end{pmatrix}, \quad f = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{K-1} \\ f_K \end{pmatrix}$$

with A_k, B_k, C_k, $k = 1,\dots,K$, $\ell \times \ell$ matrices and
x_k, f_k, $k = 1,\dots,K$, ℓ-component column vectors
(In the present case, ℓ is equal 4).

The Gauss elimination method reduces the block-tridiagonal matrix A to the product of a lower matrix L and an upper matrix U, i.e.

$$A = LU$$

where

$$L = \begin{pmatrix} L_1 & & & & & \\ B_2 & L_2 & & & & \\ & B_3 & L_3 & & & \\ & & \ddots & \ddots & & \\ & & & B_{K-1} & L_{K-1} & \\ & & & & B_K & L_K \end{pmatrix}, \quad U = \begin{pmatrix} I & U_1 & & & & \\ & I & U_2 & & & \\ & & I & U_3 & & \\ & & & \ddots & \ddots & \\ & & & & I & U_{K-1} \\ & & & & & I \end{pmatrix}$$

Solving the triangular linear systems Sy = f and Ux = y, the Richtmyer algorithm then takes the form:

$$\begin{aligned} &L_1 = A_1 \quad , U_1 = L_1^{-1}C_1, y_1 = L_1^{-1}f_1 \\ &L_k = A_k - B_kU_{k-1}, U_k = L_k^{-1}C_k, y_k = L_k^{-1}(f_k - B_ky_{k-1}) \ , \\ (2) \qquad & \qquad k = 2,\dots,K-1 \\ &L_K = A_K - B_KU_{K-1}, \qquad y_K = L_K^{-1}(f_K - B_Ky_{K-1}) \\ &x_K = y_K \\ &x_k = y_k - U_kx_{k+1}, \qquad k = K-1,\dots,1 \ . \end{aligned}$$

The lxl matrix Uk, k = 1,...,K-1, and the l-component column vectors Yk, k = 1,...K, are computed by applying the conventional Gauss elimination method to solve the l linear systems

Lk Uk = Ck for each k = 1,...,K-1

and the linear system

$$L_1\ y_1 = f_1 \quad \text{and}$$

$$L_k\ y_k = f_k - B_k\ y_{k-1} \text{ for each } k = 2,\dots,K$$

As the LU decomposition of each Lk , k = 1,...,K, and the solution of the corresponding linear systems contain recursions, and as

$$\begin{aligned} U_k &= (A_k - B_k\ U_{k-1})^{-1}C_k, && k = 2,\dots,K-1, \text{ and} \\ y_k &= L_k^{-1}\ (f_k - B_k\ y_{k-1}) && k = 2,\dots,K \\ x_k &= y_k - U_k\ x_{k+1}, && k = K-1,\dots,1, \end{aligned}$$

are defined recursively, the resulting dependences preclude the vectorization of the Richtmyer algorithm.

(b) Simultaneous Richtmyer Algorithm:

Among various possibilities, the simultaneous treatment of the Thomas algorithm, i.e. the Richtmyer algorithm for "l" equal 1, was found in general to be the most efficient algorithm for the solution of tridiagonal linear systems on the CRAY-1S and the CYBER 205 ([37],[49] in chapter 5.10, cf. chapters 5.4 and 6.3). Carrying this result over to block-tridiagonal linear systems where "l" is greater than one, only the simultaneous treatment of the Richtmyer algorithm will be considered here [84].

Since in the ξ-sweep of the Beam and Warming scheme, the block-tridiagonal linear systems for determining $\delta q*^n$ on lines of constant η are independent of each other, the dependencies stated above can be removed, if the Richtmyer algorithm is applied to these systems simultaneously by sweeping along the lines of constant ξ. Analogously, the block-tridiagonal linear systems of the η-sweep may be solved simultaneously by sweeping along the lines of constant η.

For the solution of M block-tridiagonal linear systems

$$Am\ Xm = fm\ ,\quad m = 1,\dots,M,$$

which are mutually independent, the simultaneous Richtmyer algorithm may be expressed as follows:

$k = 1$

$$L_{k,m} = A_{k,m} \qquad , \quad m = 1,\dots,M\ ,$$

$$U_{k,m} = L^{-1}_{k,m} C_{k,m} \qquad , \quad m = 1,\dots,M\ ,$$

(3)
$$y_{k,m} = L^{-1}_{k,m} f_{k,m} \qquad , \quad m = 1,\dots,M\ .$$

$k = 2,\dots,K-1$

$$L_{k,m} = A_{k,m} - B_{k,m} U_{k-1,m} \qquad , \quad m = 1,\dots,M\ ,$$

$$U_{k,m} = L^{-1}_{k,m} C_{k,m} \qquad , \quad m = 1,\dots,M\ ,$$

$$y_{k,m} = L^{-1}_{k,m} (f_{k,m} - B_{k,m} y_{k-1,m}) \qquad , \quad m = 1,\dots,M\ .$$

$k = K$

$$L_{k,m} = A_{k,m} - B_{k,m} U_{k-1,m} \qquad , \quad m = 1,\dots,M\ ,$$

$$y_{k,m} = L^{-1}_{k,m} (f_{k,m} - B_{k,m} y_{k-1,m}) \qquad , \quad m = 1,\dots,M\ ,$$

$$x_{k,m} = y_{k,m} \qquad , \quad m = 1,\dots,M\ .$$

$k = K-1,\dots,1$

$$x_{k,m} = y_{k,m} - U_{k,m} x_{k-1,m} \qquad , \quad m = 1,\dots,M\ .$$

Each of the componentwise scalar operations in (2) becomes a vector operation with a vector length of M in (3). Compared with the Richtmyer algorithm applied to M block-tridiagonal linear systems separately, the operation count is the same for the simultaneous Richtmyer algorithm. Considering the storage requirements,

3*K*l*l	floating point words for Ak, Bk, Ck,	k = 1,...K,
K*l	floating point words for fk ,	k = 1,...K,
(l*l)+l	floating point words for the solution of the linear systems to determine Uk and Yk,	

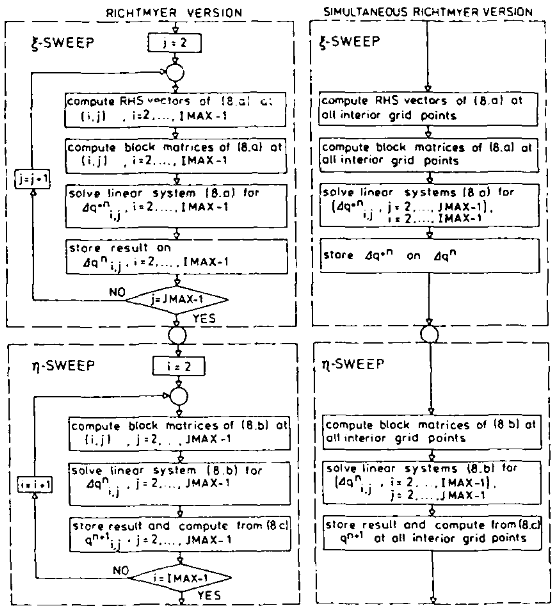
i.e. 52xK+20 floating point words in all for "l" equal 4, are needed for the Richtmyer algorithm. For the simultaneous Richtmyer algorithm the amount of storage required is increased by a factor of M.

Thus by the choice of the Richtmyer algorithm, the minimum number of arithmetic operations, and, by virtue of its simultaneous treatment, the maximum vector performance, are obtained at the cost of increased storage requirements. This may preclude the application of the simultaneous Richtmyer algorithm, where there is insufficient memory.

ADVANTAGES OF VECTORIZATION

Two versions of the blunt body code [85] for the solution of the axisymmetric thin-layer Navier-Stokes equations by the Beam and Warming scheme are compared in the following table. In the Richtmyer version, all of the most time consuming subroutines, except the one solving block-tridiagonal linear systems by the Richtmyer algorithm, are vectorized. Moreover the idea of simultaneously applying the Richtmyer algorithm is used in the simultaneous Richtmyer version.

Figure 6.6: Flow charts for the ξ- and η-sweeps of the Richtmyer and of the simultaneous Richtmyer versions (RHS = right hand side).


RICHTMYER VERSION
SIMULTANEOUS RICHTMYER VERSION
ξ-SWEEP
j = 2
compute RHS vectors of (8.a) at (i,j) , i=2,...,IMAX-1
compute block matrices of (8.a) at (i,j) , i=2,...,IMAX-1
j=j+1
solve linear system (8.a) for $\Delta q^{*n}_{i,j}$, i=2,...,IMAX-1
store result on $\Delta q^n_{i,j}$, i=2,...,IMAX-1
NO
j=JMAX-1
YES
ξ-SWEEP
compute RHS vectors of (8.a) at all interior grid points
compute block matrices of (8.a) at all interior grid points
solve linear systems (8.a) for ($\Delta q^{*n}_{i,j}$, j=2,...,JMAX-1), i=2,...,IMAX-1
store Δq^{*n} on Δq^n
η-SWEEP
i = 2
compute block matrices of (8.b) at (i,j) , j=2,...,JMAX-1
i=i+1
solve linear system (8.b) for $\Delta q^n_{i,j}$, j=2,...,JMAX-1
store result and compute from (8.c) $q^{n+1}_{i,j}$, j=2,...,JMAX-1
NO
i=IMAX-1
YES
η-SWEEP
compute block matrices of (8.b) at all interior grid points
solve linear systems (8.b) for ($\Delta q^n_{i,j}$, i=2,...,IMAX-1), j=2,...,JMAX-1
store result and compute from (8.c) q^{n+1} at all interior grid points

Considering first the common subroutines of both versions, it is obvious that the computing time for input, grid generation, and initialization is almost negligible. It is interesting to realize that the grid movement, i.e. updating the positions and the metric terms of the grid points, takes only little more time than the boundary treatment, and even less than the rather costly check of convergence. No further attempt was made to speed-up the extensive output, as only a small overall improvement could be expected.

Table 6. 6 : CRAY-1S CPU-times of Richtmyer and simultaneous Richtmyer versions of the blunt body code for 400 time levels on a 26x31 grid.

	RICHTMYER VERSION (sec)		SIMULTANEOUS RICHTMYER VERSION (sec)	
INPUT	0.01		0.01	
GRID GENERATION	0.02		0.02	
INITIALIZATION	0.03		0.03	
GRID MOVEMENT	0.40		0.40	
ξ-SWEEP	11.88		7.59	
RHS		3.73		3.56
MATRICES		1.25		1.16
RICHTMYER		6.83		2.82
STORE		0.07		0.05
η-SWEEP	11.16		7.20	
MATRICES		4.20		4.11
RICHTMYER		6.86		3.01
STORE		0.10		0.08
BOUNDARY TREATMENT	0.29		0.29	
CONVERGENCE	0.54		0.54	
OUTPUT	1.91		1.91	
TOTAL	26.24		17.99	

The differences between the Richtmyer and the simultaneous Richtmyer versions are reflected by the execution times of the ξ - and η -sweeps. For the calculation of the block-tridiagonal matrices and the right hand sides, the simultaneous Richtmyer version takes a factor of 1.04 less time than the Richtmyer version, as the number of CALL statements is reduced and the implicit treatment of the symmetry and outflow boundary conditions becomes vectorizable. But the main reason for the reduction of the total CPU-time by a factor of 1.46 is due to the vectorization of the block-tridiagonal system solver. Compared with the conventional Richtmyer algorithm, the simultaneous Richtmyer algorithm attains speed-up factors of 2.42 and 2.28 for the solution of the linear systems of the ξ -sweep and the η -sweep, respectively. The corresponding computing speeds of 35 and 33 MFLOPS on the CRAY-1S for vector lengths of 29 and 24, resp., can still be increased for longer vectors.

***Table 6. 7 : CPU-times in seconds per time level and per grid point for the blunt body code* [85].**

	Richtmyer Version	Simultaneous Richtmyer Version
IBM 3081K	$101.5 * 10^{-5}$	$120.8 * 10^{-5}$
CRAY-1S	$8.1 * 10^{-5}$	$5.6 * 10^{-5}$

Considering the implementation of the blunt body code on the CYBER 205, the advantages of the simultaneous Richtmyer version will carry over. Compared with the CRAY-1S, a higher performance for forming the block-tridiagonal linear systems can be expected, because long, contiguously stored vectors can be used to take advantage of the higher peak MFLOPS rate of the CYBER 205. But for the simultaneous solution of the systems (cf. chapter 6.3 for the simultaneous treatment of the Thomas-algorithm) and also for the calculation of the systems in the Richtmyer version of the program, not all of the vectors are contiguously stored, and the vector lengths are small. Therefore the high vector performance of the CYBER 205 can only be exploited to a small degree by the implicit Beam and Warming scheme.

Since longer vectors can be formed and arrays instead of vectors can be used, resp., the Beam and Warming scheme is expected to be more efficient on a vector computer like the ETA-10 and a parallel processor like the former Illiac IV [94], resp., for three-dimensional time-dependent problems than for two-dimensional ones, provided there is sufficient memory.

To estimate the gain of performance for the CRAY-1S, the Richtmyer and the simultaneous Richtmyer versions of the blunt body code were run on the IBM 3081K using the AUTODBL compiler option and with the CRAY-1 FORTRAN intrinsic functions ISMAX and SSUM replaced by standard FORTRAN statements. The vectorization of the original version of the program proved to be profitable on the IBM 3081K as well. The Richtmyer version led to a reduction of CPU-time by a factor of 1.37. Because of increased paging, the simultaneous Richtmyer version reached a factor of only 1.15.

Comparing the Richtmyer version on the IBM 3081K with the simultaneous Richtmyer version on the CRAY-1S, the CPU-time per grid point and per time level was reduced from 101.5 x 10E-5 sec to 5.6 x 10E-5 sec, corresponding to a speed-up factor of 18.19.

As the Richtmyer algorithm cannot fully exploit the high vector performance of the CRAY-1S, there is still need for a vectorizable block-tridiagonal system solver with little storage requirements, if the available storage prohibits the application of the simultaneous Richtmyer algorithm.

Table 6.8 : Speed-up factors of the Richtmyer Version (RV) and the Simultaneous Richtmyer Version (SRV) of the blunt body code.

RV on IBM 3081K / RV on CRAY-1S	SRV on IBM 3081K / SRV on CRAY-1S	RV on IBM 3081K / SRV on CRAY-1S
12.47	21.66	18.19

6.7 A NOTE ON UNSTRUCTURED GRIDS

The application of mesh generation techniques to complicated geometries becomes increasingly difficult when one tackles three-dimensional problems. Considerable ingenuity is required to form a network of cells that are not too distorted and yet meet all the conditions previously mentioned. Successful mesh generation methods that use cube-like cells have been developed for wing-body combinations and for a combination of wing, body, and tail. However, it becomes increasingly difficult to keep boundary surfaces aligned with cell faces when one uses a regular structure of rectilinear cells. This difficulty has hindered the development of flowfield computational methods to treat a complete aircraft including engine nacelles and struts.

An alternative to the mesh formed by an array of rectilinear cells is the use of triangular elements in two dimensions or tetrahedra in three dimensions [4]. With this cell type there is no longer any need to retain structure in the mesh. Indeed the lack of any natural coordinate direction or need for structure becomes a virtue because it is always possible to connect a set of points to form a covering of triangles in two dimensions or tetrahedra in three dimensions. For the airplane code [53] a method has been developed for calculating inviscid transonic flow over a complete aircraft based on an unstructured mesh of tetrahedra.

The present version requires eight million words and takes about one hour to run on a CRAY X-MP/48 Computer. Approximately one-third of this time is consumed by the triangulation procedure that automatically connects the points to form the tetrahedra. The remaining time is taken up by the solution algorithm that calculates the flowfield. The present mesh contains 20,000 points and is made up of 100,000 tetrahedra. This is not sufficient for an accurate solution, and realistic calculations will require many more points than are now used. Memory requirement for an acceptably accurate result will be about 50 million words.

The method, which is based on an unstructured mesh of cells, differs from most fluid dynamics codes currently in use or under development, and is more closely related to finite elements methods used in structural analysis. It poses a variety of novel problems, both in the triangulation process and in the design of a suitable algorithm. Methods based on a structured network of rectilinear cells usually lend themselves readily to vectorization. For example, current wing and wing-body-tail codes can sustain a rate of about 70 MFLOPS on a CRAY X-MP computer system. For the unstructured aircraft code the MFLOP rate drops to about one-tenth of this speed unless steps are taken to achieve a vectorizable algorithm.

Suppose that the cells are labeled by the parameter L = 1,NCELL. In two dimensions each cell is a triangle with three vertices. Each vertex has a position defined by its x and y coordinates. Let the n-th point have coordinates x(n,1) and x(n,2). A typical part of the flow algorithm might require a loop over the cells in which the coordinates of the cell vertices are required in some further computation. Thus, we may suppose that we have an array, say NDC(L,I) where L refers to the cell number and I = 1,2,3, refers to the three vertices. Thus, a typical loop might have the form

```
    DO 100 L = 1,NCELL
    N1 = NDC(L,1)
    N2 = NDC(L,2)
    N3 = NDC(L,3)
    X1 = X(N1,1)
    Y1 = X(N1,2)
    X2 = X(N2,1)
    Y2 = X(N2,2)
    X3 = X(N3,1)
    Y3 = X(N3,2)
         .
         .
         .
100 CONTINUE
```

The positions of the triangle vertices are thus defined by the coordinate pairs (X1,Y1), (X2,Y2), and (X3,Y3). However, the use of indirect addressing mandated by the unstructured nature of the mesh means that each point will appear at least three times because it will be referenced as a vertex of at least three different triangles. The possibility of a vector dependency will inhibit vectorization. However, if one first sorts the cells into groups so that no vertex is referenced more than once in each group, one can override the compiler and force vectorization confident that vector dependency will not occur. Hardware gather and scatter operations then ensure that the vectorized algorithm for an unstructured mesh will have a processing rate comparable to that achieved by the traditional algorithms which do not require indirect addressing. The loop now takes the form

```
    DO 110  K = 1,KGRP
    L1 = LGRP(K)
    L2 = LGRP(K+1)-1
    DO 100  L = L1,L2
         .
         .
         .
100 CONTINUE
110 CONTINUE
```

The sorting of a triangulated region into disjoint groups of triangles such that no vertex occurs more than once in each group is reminiscent of map coloring problems. One might therefore expect that ingenious sorting methods would generate the smallest number of possible groupings. A naive sorting algorithm is already in use and can achieve a fivefold improvement over a straight scalar computation, resulting in a sustained processing rate of 20 to 40 MFLOPS. The variation in processing rate is caused by the variation in the number of cell groups, which depends on the mesh. One might expect improvements in sorting algorithms to lead to improved MFLOP rates.

6.8 VECTORIZATION AND MULTITASKING OF A MULTI-GRID ALGORITHM

The adaptation of multi-grid methods to the computation of Euler and Navier-Stokes flows has been one notable trend in CFD since the early 80's. At present, multi-grid algorithms are being used to accelerate the convergence of steady flow simulations. However, it appears that their utility may extend to the time-accurate computation of unsteady flows. For the following discussions on vectorization and multitasking we closely follow the description in the reports by Johnson et als. [54], [55].

The introduction of multi-grid methods preceded the arrival of modern concurrent processors. Consequently, their design has typically been based on the sort of sequential, scalar reasoning appropriate for SISD machines. Restructuring multi-grid algorithms is therefore necessary with respect to the special architectures.

There are now available a variety of explicit and implicit sequential multigrid algorithms for the solution of systems of conservation laws of the form

$$Q_t = -(F_x + G_y)$$

with the conservation vector Q. This equation may, for example, represent the Euler equations, the full Navier-Stokes equations or the thin-layer Navier-Stokes equations, depending on the choice of the vector Q, F and G.

The multi-grid algorithm consists of a fine-grid solution procedure and a coarse-grid acceleration scheme. The fine-grid procedure solves the unsteady equations of motion and may, if desired, do so in a time-accurate manner. A variety of implicit and explicit methods may be used to construct the fine-grid procedure. Here, because of its simplicity and ubiquity in computational aerodynamics, we choose to use the explicit, two-step Lax-Wendroff scheme known as MacCormack's method [77]. The fine grid is constructed such that the number of points in each direction is expressible as n(2**p)+1 for p and n integers such that $p \geq 0$ and $n \geq 2$, where p is the number of grid coarsenings and n is the number of coarsest-grid intervals. A collection of successively coarser grids is then created by a recursive process which deletes every other point in each coordinate direction.

Information is transferred from the fine grid to each of the coarser grids. This transfer may be accomplished either by a sequential cascading of information through successively coarser grids or by a simultaneous communication directly from the fine grid to all of the coarser grids. In any case, a coarse-grid scheme is then used to rapidly propagate the resolvable components of this fine-grid information throughout the computational domain to accelerate convergence to the steady state while maintaining the accuracy determined by the fine-grid discretization.

In the sequential grid updating algorithm, the solution is advanced over one multiple-grid cycle as follows. First a fine-grid correction, dQ1, is computed. Then dQ1 is restricted to the next-coarser grid, where dQ2 is computed. The dQ2 correction is both restricted to grid 3 and interpolated onto grid 1, where it provides an additional update to the fine-grid solution. On grids 3 through N-1 the procedure is analogous to that on grid 2. When dQn has been computed and interpolated onto grid 1 to provide the Nth update to the fine-grid solution, the next multi-grid cycle is ready to begin.

Observe that when the components of the sequential grid updating algorithm (namely, the fine- and coarse-grid schemes) are both explicit, it is particularly easy to vectorize. However, the effectiveness of vectorizing the coarse-grid scheme is limited by the progressively shorter vectors which may be constructed on the successively coarser grids. Such an explicit sequential algorithm may also be run on an MIMD machine by splitting each grid, in turn, across the total number of processors available. An implicit sequential grid updating algorithm would probably vectorize less well and also require additional redesign to run on a parallel processor.

The parallel coarse-grid algorithm removes the dependence of grids 3 through N upon their immediate predecessors. In particular, dQ1 is now restricted to each of grids 2 through N. All of these coarse grids may then be updated simultaneously and independently of each other.

This allows the mesh points on grids 2 through N to be assembled into one vector in order to improve performance on an SIMD computer. Alternatively, the coarse grids could each be updated simultaneously on separate processors of an MIMD machine. This would be attractive, for example, if the coarse-grid scheme were implicit.

A further possibility is a fully parallel algorithm. Here dQ1 from the previous cycle is restricted to each of the coarse grids. This makes all of the grids 1 through N independent of each other and allows their simultaneous update.

VECTORIZATION

As both the fine-grid solution procedure and the coarse-grid acceleration schemes used in [54,55] are explicit, the resultant multi-grid algorithms are readily vectorizable. Such vectorization of the sequential algorithm has been performed for computation on a CDC CYBER 205. First the code was rewritten to take full advantage of the automatic vectorization which is performed by the CYBER 205 compiler. For the conservation vector Q and the flux vectors F and G four quantities must be computed at every point in the two-dimensional domain. These vectors are stored in three-dimensional arrays. The array indices were arranged in decreasing length from left to right, and the DO loops containing these indices were likewise ordered so that the innermost loop corresponds to the longest dimension (i.e., the first index), the second innermost loop corresponds to the next longest dimension (the second index), and so on. These modifications enable access to contiguous locations in the vectors so that the loops automatically vectorize. When nested DO loops result in access to every point in the two-dimensional domain, including the boundaries, entire matrices are treated as single long vectors containing the whole flowfield. With these changes, the code compiled with the automatic vectorization option ran approximately 2 to 4 times faster than the code using only scalar optimization [55].

Further vector speedup was obtained by implementing CYBER 205 explicit vector FORTRAN. Bit vectors were created for use in WHERE blocks to control storage for vectorized computations that involved only the interior points of the domain. Dynamic storage was introduced so that temporary vectors could be used to reduce the operations count. Vector intrinsic functions (such as Q8VGATHR, Q8VCMPRS, etc.) were used to build contiguous vectors from the array elements needed on the coarse grids.

The parallel coarse-grid algorithm has been vectorized in a similar fashion, with the additional feature of combining the points to be updated on grids 2 through N into one long vector. This minimizes the algorithm on an SIMD computer.

MULTITASKING

When attempting to multitask an algorithm for execution on an MIMD machine, we are concerned with multitasking overhead and algorithm granularity. By granularity we mean the time required to execute a multitaskable segment of the algorithm on a single processor. For a given multitasking overhead, the best speedup is obtained when algorithm granularity is maximal. Large granularity is usually introduced by top-down programming which exploits global parallelism in the algorithm. Bottom-up programming, on the other hand, exploits algorithm parallelism at a low level by making many partitionings, each on small code segments, such as DO loops containing independent statements.

The sequential multigrid algorithm contains many opportunities for creating small granularity parallelism but relatively few opportunities for the sort of large granularity necessary to produce good multitasking speedup in the face of non-trivial multitasking overhead. This observation, together with the desirability of non-sequential multigrid schemes for reasons of algorithm flexibility, led to the construction of the parallel multigrid algorithms described above. In these algorithms, grids which are independent of one another may be updated simultaneously on separate processors. In fact, such a simple strategy may result in a poor load balance across processors because of the different amounts of work inherent in updating grids of different coarseness. However, more refined strategies are possible. Grids may, for example, be grouped together into tasks of approximately equal work, or they may be melted into tasks with other large-grained multitaskable code segments in order to equilibrate processor loading.

The scalar sequential algorithm in [55] yields multi-grid speedups of 6.9, 2.9 and 8.2 for selected inviscid supercritical and turbulent viscous flows, respectively. Explicit vectorization of this algorithm results in vectorization speedups in the vicinity of 3.0, 2.9 and 2.7 for the respective cases cited above.

The parallel coarse-grid algorithm maintains essentially the same convergence behavior as the sequential algorithm. Consequently, the multi-grid speedups obtained with it are virtually identical to those of the sequential algorithm. Vectorization of the parallel coarse-grid algorithm yields speedups of 3.2, 3.0 and 2.8 for the three test cases considered in [55].

The theoretical maximum speedup on a p-processor MIMD machine is p. Varying overhead requirements of the multiple grid algorithms will obviously result in distinct actual multitasking speedups. Using a top-down multitasking approach, the parallel coarse-grid algorithm has been implemented on a four processor CRAY X-MP and on a Denelcor HEP I. Initially, only the coarse grids were multitasked so that the performance of parallel grids on a multiprocessor could be evaluated. Then the fine-grid computations were partitioned and multitasked, and the resultant code was integrated with the parallelized coarse grids. Load balancing of the entire scheme completed the study of performance resulting from the top-down approach. Multitasking results are reported in [55].

6.9 VECTORIZATION OF A GALERKIN METHOD

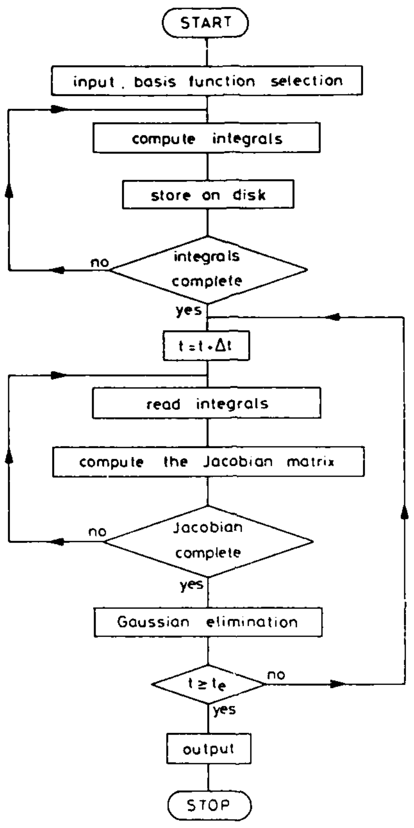
For the numerical simulation of incompressible flows spectral methods were used for many applications. This class of numerical methods has particular advantages in solving instability and transition problems.

A special type of spectral methods is the Galerkin method. To illustrate this method and discuss vectorization [58], we examine the instabilities of a convective flow which is confined to a rectangular box heated from below. To simulate these instabilities, the three-dimensional, time-dependent Boussinesq equations were employed. The main interest of the numerical simulation is to calculate the time-dependent instabilities which occur in such a flow. The results of these calculations using the Galerkin method were given in [58]. In order to solve the basic equations with the Galerkin method we must expand the unknown functions (velocity v, temperature T) in sets of given basis functions. The unknown coefficients of the basis functions were determined in such a way that the error is minimized in the computational region. The criterion for minimizing the error function gives the equations for the coefficients.

To solve these ordinary differential equations the time derivatives were replaced by a finite-difference operator. As the equation can be very stiff, explicit methods tend to numerical oscillations especially for large systems of equations. We therefore employ a derivation of the trapezoidal rule, the so-called "One-leg method."

This method is implicit, of second-order accuracy and remains stable up to large time steps. The Galerkin equations now were reduced to a system of non-linear algebraic equations for the unknown coefficients of the new time level. These non-linear equations were solved using a Newton method. The structure of the Jacobian matrix depends strongly on the basis functions chosen, and the values of the Rayleigh number and Prandtl number. As the matrix is full and the values of the diagonal are not dominant, an iterative method is not efficient to solve the linear systems. We therefore use the Gaussian elimination.

Figure 6. 7 : Flow chart of the Galerkin code.



The implementation of the Galerkin method is more complicated compared to corresponding finite difference schemes. Thus we confine ourselves to a discussion of the principle structure of the program. It can be divided into three main parts:

1. Calculation of the integrals.
2. Calculation of the Jacobian matrix.
3. Solving the linear system.

The number and the values of the integrals only depend on the basis functions. They remain constant during the whole calculation. Therefore there are two possibilities for the structure of the program.

The first version is to calculate the integrals at each time step simultaneously with the calculation of the Jacobian matrix. Thereby reducing the storage requirement. Most of the storage is needed then for the Jacobian matrix. The total storage requirement is about 1.4*N**2, where N is the number of basis functions. To simulate the 3D time-dependent convection flow 400 functions are necessary corresponding to a relatively small storage of 225,000 words.

In the second version the integrals are calculated once and stored before the first time step. As the integrals are required at each time step, this saves a lot of computation time. According to the type of computer, the computation time decreases by factors of about 2.5 to 7. In contrast to this advantage, the requirement of storage increases enormously. The number of integrals depends on the number of basis functions and the selection modes used. For 400 functions, about 7 to 8 million words have to be stored. In the case the memory is not large enough, say one million words, the values of the integrals have to be written on disk in blocks of about 270000 words. The data management is very easy, because every integral is required only once per time step.

In [58] the second version of the Galerkin method has been realized. The principle parts of the program are explained with the flow chart shown in Fig.6.13. After the input of all parameters the integrals FINT were calculated in the same sequence as they were required later. After reaching the limit of 270000 the array FINT will be stored on disk and the next integrals can be calculated.

This operation will be repeated until all required integrals are calculated and stored on disk. Setting n equal to 1 the first time step begins. To determine the Jacobian matrix the values of the integrals are restored from disk to the array FINT again in blocks of 270000 words. After the Jacobian has been completely calculated the linear system is solved employing the Gaussian elimination, and the next time steps can be calculated until the final time TEND is reached.

To vectorize the code efficiently, the compiler option ON = F has been employed giving information about the time required by each subroutine during the execution of the program. The result of this is summarized in Table 6.14.

Table 6.9 : CRAY-1S CPU-time (in seconds) for the original version of the Galerkin code, 398 basis functions.

Input, Output, basic function selection	1.60	13.78
Integrals, calculation and store	12.18	start-up time
Jacobian matrix	6.82	
Subroutine DOVEK	5.47	
read integrals	1.30	9.92
rest	0.05	per time step
Gaussian elimination	3.10	

The input of parameters, the selection modes for the basis functions and all output utilities together require 1.60 seconds assuming a number of 398 basis functions. Before executing the first time step, one has to calculate all the integrals and to store them on disk. This part of the program requires 12.18 seconds. The code to calculate the Jacobian matrix is well structured and most of the time is spent in the subroutine DOVEK which contains only some nested DO-loops. Including the time for reading the integrals from disk, the total time required is 6.82 seconds. 3.10 seconds are needed for solving the linear system by Gaussian elimination.

In the sections of the code containing the selection of the basis functions and the calculation of the integrals, vectorization is very difficult. The selection modes for the basis functions is held very variable and there is no chance to get a good performance in these parts of the code. When being interested mainly in time-dependent calculations with the need of 500 to 5000 time steps, the start-up time can be neglected compared to the time required for all time steps. Therefore vectorization of the subroutine DOVEK and the Gaussian elimination is most important.

As discussed above the integrals stored on disk have to be read in blocks of about 270000 words at each time step. In the example shown in Table 6.14 total number of integrals is 7.3 * 10**6. Using an implicit DO-loop in the READ-statement, which is vectorized on the CRAY-1S, the CPU requires 1.30 seconds to read the whole data. Replacing the implicit DO-loop by the BUFFER IN statement the time to read the data is reduced by a factor of 65 in this case. The 0.02 seconds now required to transfer the data can be neglected compared to the other operations per time step. Using this fast data transfer, one is no longer restricted by the relatively small memory of the CRAY-1S.

To solve the linear system defined by the Jacobian matrix a vectorized Gaussian elimination discussed in [40] has been employed leading to a reduction in the computational time by a factor of up to 3.8. Considering the calculation with 398 basis functions the elimination process now requires 0.81 seconds. Assuming 0.66*N**3 floating point operations to solve the N linear equations, a computing speed of 52 MFLOPS for N equal to 398 has been obtained [59].

As the Galerkin equations consist of sums of linear and quadratic terms only, the parts of the derivatives resulting from a linear term lead to a single loop, which is already vectorized. To calculate the parts resulting from the quadratic terms, two nested loops are required. These loops contain more than 98.5 per cent of the operations needed to build up the Jacobian matrix, and are integrated in the subroutine DOVEK. The whole task now is to vectorize this one subroutine.

In the original version the subroutine DOVEK contained the following nested loops

```
      DO   1    I = IA,IE
      DO   1    J = JA,JE
      NB = NB + 1
      PA(I) = PA(I) + X(J) x FINT(NB)
    1 PA(J) = PA(J) + X(I) x FINT(NB)
```

The partial derivatives of one equation are stored in the array PA and the coefficients in the array X. To improve the performance of these loops we do the following steps:

a) Remove the dependences:

Vectorization of the inner loop yields wrong results if I is equal to J. Therefore the loop is not vectorized by the autovectorizer. To avoid the dependences the inner loop is split into two loops. As the inner loops are now vectorized the performance improves by a factor of about 3.4.

b) Increase length of inner loops:

As the basis functions of the velocity vector and the temperature are composed of 7 subsystems of different symmetry, the length of the inner loops varies from about 20 up to 100 even if all 400 functions are used. Defining a new array X1, we can copy two or three parts of the vector X to the new array to store the relevant parts contiguously. Processing the longer vector X1, one can avoid especially the very short loops. Although one has to copy the vectors each time before processing the nested loops a speed up factor of 1.3 has been obtained.

c) Use the CRAY-1S intrinsic functions:

Considering the nested DO-loops in the above example one can recognize the different vector structures of the two statements in the inner loop. As the index I of the outer loop has a constant value for the inner loop, the linear expressions have the form

```
SCALAR = SCALAR + VECTOR * VECTOR
VECTOR = VECTOR + SCALAR * VECTOR
```

Therefore the expressions can be replaced by the intrinsic functions SDOT, which calculates the dot-product of two vectors and the function SAXPY which adds a vector and another scaled one. The improvement of performance by employing these functions is not very high. Whereas the function SDOT has a speed-up factor of greater than one for all vector lengths, SAXPY is faster than the corresponding DO-loop only for a vector length greater than about 100.

Although nearly all operations to calculate the Jacobian matrix are done using intrinsic functions, and therefore more than 99 per cent of this part of the code is vectorized, the performance is relatively poor. Employing the example with 398 basis functions, one can obtain a rate of only 30 MFLOPS. The maximum perrormance of more than 60 MFLOPS is attained only for a vector length of more than 300 for SAXPY and more than 1000 for SDOT. Employing loops with lengths in the range of 50 to 150 the function does not work very efficiently.

As there is no possibility to further increase the length of the vectors used within the algorithm, another change of the DO-loops is more efficient:

d) Unroll the outer loop partially:

First, one has to replace the intrinsic functions by the original inner DO-loops. To increase the number of operations in the inner loop, we are processing simultaneously the expressions for I, I+1, I+2 and 1+3. Therefore the increment of the outer loop can be changed from 1 to 4. If the expression in the inner loop is a triadic operation of the form

```
VECTOR = VECTOR + SCALAR * VECTOR
```

this modification leads to rather high performance even for short vector lengths. To illustrate the modification, we show the original nested loops in comparison to the partially unrolled outer loop:

original

```
     DO   1    I = IA,IE
     DO   1    J = JA,JE
     NB = NB+1
  1  PA(J) = PA(J) + X(I) * FINT(NB)
```

partially unrolled outer loop

```
     JD = JE+1-JA
     DO   1   I = IA,IE4
     DO   2   J = JA,JE
     NB = NB+1
   2 PA(J) = (((PA(J) + X(I) * FINT(NB)) + X(I+1) * FINT(NB+JD))
    $        + X(I+2) * FINT(NB+2*JD)) + X(I+3) * FINT(NB+3*JD)
   1 NB = NB+3*JD
```

By using the parenthesis on the right hand side of the expression in the DO-loop we enhance chaining. If (IE-IA) is not a multiple of the increment 4, more operations are processed than in the original version. To avoid errors in this case the dimension of some arrays have to be increased and a copy of the vector X is necessary before processing the loop. Nevertheless the speed-up factor compared to the fully vectorized version with intrinsic functions is more than 1.7.

To unroll the outer loop is only profitable if the expression in the inner loop is of triadic form. Therefore we want to change all dot products to the triadic form. This can easily be done by exchanging the inner and outer loop, which is illustrated by the following examples:

Dot product:

```
     DO   1   I = IA,IE
     DO   1   J = JA,JE
     NB = NB + 1
   1 PA(I) = PA(I) + X(J)*FINT(NB)
```

Triadic operation:

```
     JD = JE + 1 - JA
     DO   1     J = JA,JE
     NB = NB + 1
     II = 0
     DO   1     I = IA,IE
     II = II + 1
   1 PA(I) = PA(I) + X(J)*FINT(NB+JD*II)
```

Now we can partially unroll the outer loop again according to the previously discussed example. Although in this example the inner loop is now the short one, the performance of the partially unrolled loop is better than the version using the intrinsic function SDOT.

All nested loops in the subroutine DOVEK can now be formulated in the same manner with a partially unrolled outer loop. Employing the example with 398 basis functions, the performance improves to 51 MFLOPS for calculating the Jacobian matrix. Considering the asymptotic performance of 66 MFLOPS for the scalar product of very long vectors, this is a rather good result on the CRAY-1S.

The benefit of all modifications described above will be shown by comparing the original version and the completely vectorized version on the CRAY-1S. In addition to this, the modified version is run with the autovectorizer switched off with the option OFF = V, that means no vectorization of the inner loops. The results of this comparison are shown in Table 6.10.

Table 6.10: Comparison of original and vectorized versions, CRAY CPU-times in seconds for 398 basis functions and one time step.

	original version	vectorized version	vectorized version autovectorizer off
Input, Output basis function selection	[illegible]	[illegible]	2.37
Integrals, calculation and storage	[illegible]	[illegible]	[illegible]
Jacobian matrix	[illegible]	[illegible]	[illegible]
Gaussian elimination	3.1	[illegible]	[illegible]

As there are no modifications, the time required for input and output remains the same in the two versions. The decrease in time for calculating and storing the integrals is the profit of the BUFFER OUT statement. The time required for the calculations of the Jacobian matrix is decreased considerably by vectorization of the code. Also the Gaussian elimination used in the vectorized version has a speed-up factor of nearly four compared to the original version. By turning the autovectorizer off, we can see that only little is vectorized in the code computing the integrals and doing input and output. However, factors of 6 to 8 can be obtained by computing the Jacobian and the Gaussian elimination.

To get a feeling of the influence of the start-up time, we give a realistic example of solving the time dependent equations using 500 time steps. The CPU time and the relative importance of the sections are shown in Table 6.16.

Table 6.11: CPU-time on the CRAY-1S of the modified Galerkin code, 398 basis functions, 500 time steps (% : per cent).

	time (sec)	%
input, output	1.6	0.2
integrals	10.8	1.5
Jacobian matrix	309.6	42.5
Gaussian elimination	406.8	55.8
total	728.8	100.0

One can easily see that the start-up time required by input/output and the calculation of the integrals can be neglected, and only the highly vectorized parts of the code are important.

6.10 LITERATURE

The references presented below give only a brief overview of the understanding engineers and scientists have realized in the fields of computational fluid dynamics with special emphasis on the Navier-Stokes equations, with the aid of supercomputers during the last years. For more information the reader is referred to the more detailed literature in the special references.

[1] Abolhassani J.S., Smith R.E. and Tiwari S.N.: Numerical Solutions of Navier-Stokes Equations for a Butler Wing. AIAA Paper 87-0115, 1987.

[2] Adamczyk J.J., Graham R.W.: Numerical Simulation of Multiblade Row Turbomachinery Flows. CRAY Channels 8, 1986 No.2, 12-17.

[3] Azar A., Gaglot Y.: Vectorization of explicit multi- dimensional finite difference and finite element schemes. Proc. 1. Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983, 23-29.

[4] Baker T., Jameson A.: Computational Methods and Aerodynamics. CRAY Channels 198

[5] Barkay D., Moriarty K.J.M.: Can the Monte Carlo Method for Lattice Gauge Theory Calculations be Effectively Vectorized? Comp. Phys. Comm. 27, 1982, 105-111.

[6] Barth T.J., Pulliam T.H., Buning P.G.: Navier-Stokes Computations for Exotic Airfoils. AIAA-Paper 85-0109, 1985.

[7] Barton J.T., Pulliam T.H.: Airfoil Computation at High Angles of Attack, Inviscid and Viscous Phenomena. AIAA- Paper 84-0524, 1984.

[8] Barton J.T.: Early Experiences with the NAS CRAY-2. NASA Ames 1986.

[9] Beam R.M. and Warming R.F.: An implicit factored scheme for the compressible Navier-Stokes equations, AIAA J. 16 1978 393-402.

[10] Berendsen H.J.C., van Gunsteren W.F., Postma J.P.M.: Molecular dynamics on CRAY, CYBER and DAP. Proc. NATO Advanced Research Workshop on High-Speed Computation, Juelich, 20-22 June, 1983.

[11] Berger M., Oliger J., Rodrigue G.: Predictor-corrector methods for the solution of time-dependent parabolic problems on parallel processors. In: Elliptic Problem

[12] Book D.L.: Finite-difference techniques for vectorized fluid dynamics calculations. Springer-Verlag, New York 1981.

[13] Brandt A.: Multigrid solvers on parallel computers. In: Elliptic Problem Solvers (Schultz, M., ed.), Acad. Press New York 1981, 39-84.

[14] Bristeau M.O.: GAMM workshop on "Numerical Simulation of Compressible Navier-Stokes Flows", Dec. 4-6, 1985, Nice. To be published in "Notes on Numerical Fluid Mechanics", Vieweg Verlag.

[15] Brocard O., Bonnet C., Vigneron Y., Lejal T., Bousquet J.: A vectorized finite element method for the computation of transonic tridimesional potential flows. Proc. 1. Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983, 45-50.

[16] Butcher W.: The solution of the seismic one way equation on parallel computers. Proc. Int. Conf. "Parallel Computing 83", North-Holland Publ. 1984.

[17] Buzbee B., Golub G., Howell J.: Vectorization for the CRAY-1 of some methods for solving elliptic difference equations. In: High Speed Computer and Algorithm Organization (Kuck, D.J. Lawrie, D. H., Sameh, A.H., eds.), Acad. Press, New York 1977, 255-272.

[18] Buzbee B. L.: Implementing techniques for elliptic problems on vector processors. In: Elliptic Problem Solvers (Schultz, M., ed.), Acad. Press, New York 1981, 85-98.

[19] Candler G.V., MacCormack R.W.: Hypersonic Flow Past 3-D Configurations. AIAA Paper 87-0480, 1987.

[20] Carr M. P., Forsey C.R.: Developments in Coordinate Systems for Flow Field Problems. In: Roe, P. L. (Ed.): Numer. Meth. in Aeronautical Fluid Dynamics. Acad. Press 1982.

[21] Chaderjian N.M., Steger J.L.: A Zonal Approach for the Steady Transonic Simulation of Inviscid Rotational Flow. AIAA-Paper 83-1927, 1983.

[22] Chaderjian N.M.: Transonic Navier-Stokes Wing Solutions Using a Zonal Approach. AGARD 58th Panel Symp., Aix-en- Provence 1986.

[23] Chang J.L.C., Kowak D., Dao S.C., Rosen R.: A 3D Incompressible Flow Simulation Method and its Application to the Space Shuttle Main Engine. AIAA-Paper 85-0175, 1985.

[24] Chapman D.R.: Computational Aerodynamics Development and Outlook. AIAA Journal 17 1979 No 12, pp. 1293-1313.

[25] Chima R.V., Johnson G.M.: Efficient solution of the Euler and Navier-Stokes equations with a vectorized multiple- grid algorithm. AIAA-Paper 83-1893 1983.

[26] Current Capabilities and Future Directions in CFD. Aeronautics and Space Eng. Board, NRC 1986.

[27] Doughery F.C., Benek J.A., Steger J.L.: On Applications of Chimera Grid Schemes to Store Separation. NASA TM 88193, 1985.

[28] Edwards J.W., Thomas J.L.: Computational Methods for Unsteady Transonic Flows. AIAA Paper 87-0107, 1987.

[29] Ergel J., Lichnewsky A., Thomasset F.: Parallelism in finite element computation. Proc. IBM Symp. on Vector Computers and Sc. Comp., Rome 1982.

[30] Evans D.J.: The parallel solution of partial differential equations. Proc.Int. Conf."Parallel Computing 83", North-Holland Publ. 1984.

[31] Flores J.: Convergence Acceleration for a Three- Dimensional Euler/Navier-Stokes Zonal Approach. AIAA Paper 85-1495, 1985.

[32] Flores J., Holst T.L., Kaynak U., Gundy K., Thomas S.D.: Transonic Navier-Stokes Wing Solution Using a Zonal Approach. Part 2. AGARD Conf. Proc. 412, 1985.

[33] Flores J., Holst T.L., Kaynak U., Gundy K., Thomas S.D.: Transonic Navier-Stokes Wing Solution Using a Zonal Approach. Part 1. NASA TM 3248, 1986.

[34] Flores J., Reznick S.G., Holst T.L., Gundy K.: Transonic Navier-Stokes Solutions for a Fighter-Like Configuration. AIAA-Paper 87-0032, 1987.

[35] Fujii K., Obayashi S.: Navier-Stokes Simulation of Transonic Flow over Wing-Fuselage Combinations. AIAA Paper 86-1831, 1986.

[36] Gentzsch W.: High performance processing needs in fluid dynamics. Proc. SEAS Spring Meeting, Amsterdam 1982, 575- 590.

[37] Gentzsch W.: Benchmark results on physical flow problems. Proc. Conf. High-Speed Comput., Juelich 1983.

[38] Gentzsch W.: Numerical Algorithms in CFD on Vector Computers. Parallel Computing 1, 1984, 19-33.

[39] Gentzsch W. (Ed.): Finite Volume Methods for the Potential, Euler and Navier-Stokes Equations. DFVLR-IB 221-84 A 10, 1984

[40] Gentzsch W.: Vectorization of Computer Programs with Applications to Computational Fluid Dynamics. Vieweg Publ. Comp., Braunschweig F.R.G. 1984.

[41] Gentzsch W.: The Optimal Use of Vector Computers in Computational Physics. Proc. of the SEAS Spring Conferecne 1985.

[42] Gentzsch W.: Benchmark Results for the IBM 3090-200 VF, UNISYS ISP, FUJITSU VP 200, CRAY X-MP and CRAY-2. PIC, Vol. 10, No 2, 1987 (in german).

[43] Gentzsch W.: Benchmark results for the CONVEX C-1, ALLIANT FX/8 and SCS-40. PIC, Vol. 10, No 3, 1987 (in german).

[44] Gnoffo P.A., McCandless R.S., Yee H.C.: Enhancements to Program LAURA for Computation of Three-Dimensional Hypersonic Flow. AIAA Paper 87-0280, 1987.

[45] Gregg R.D., Misegades K.P.: Transonic Wing Optimization Using Evolution Theory. AIAA 1987.

[46] Haase W., Echtle H.: Computational Results for Viscous Transonic Flows Around Airfoils. AIAA Paper 87-422, 1987.

[47] Hankey W.L., Graham J.E., Shang J.S.: Navier-Stokes solution of a slender body of revolution at large incidence. AIAA-Paper 81-0190 1981.

[48] Hemker, P.W., Wessling P., de Zeeuw, P.M.: A Portable Vector-Code for Autonomous Multigrid Modules. In: PDE Software (Eds. B. Engquist, T. Smedsaas) Elsevier Publ. 1984.

[49] Hodous M.F., Bozek D.G., Ciarelli D.M., Ciarelli K.J., Kline K.A., Katnik R.B.: Vector processing applied to boundary element algorithms on the CDC CYBER 205. Proc. 1. Int. Coll. on Vector and Parallel Computing in Scient. Appl., Bulletin de la Direction des Etudes et Recherches, Serie C, 1 1983, 87-94.

[50] Holst T.L., Thomas S.D., Kaynak U., Gundy K.L., Flores J., Chaderjian N.M.: Computational Aspect of Zonal Algorithms for Solving the Compressible Navier-Stokes Equations in 3D. NASA TM 86774, 1985.

[51] Holst T.L.: AIAA "Viscous Transonic Airfoil" Workshop, AIAA 25th Aerospace Sciences Meeting, Reno, Januray 1987.

[52] Jameson A., Schmidt W., Turkel E.: "Numerical Solutions of the Euler Equations by Finite Volume Methods Using Runge- Kutta Time-Stepping Schemes", AIAA 81-1259, June 1981.

[53] Jameson A., Baker T.J.: Improvements to the Aircraft Euler Method. AIAA-Paper 87-0452, 1987.

[54] Johnson G.M., Swisshelm J.M.: Multiple-Grid Solution of the 3D Euler and Navier-Stokes Equations. ICS Techn. Report 84001, 1984.

[55] Johnson G.M., Swisshelm J.M.: Concurrent-Processing Adaptations of a Multiple-Grid Algorithm. ICS Techn. Report 86001, 1986.

[56] Keller J.D., Jameson A.: Preliminary study of the use of the STAR-100 computer for transonic flow calculations. AIAA-Paper 78-12 1978.

[57] Kenichi Matsuno: A vector-oriented finite-difference scheme for calculating three-dimensional compressible laminar and turbulent boundary layers on practical wing configurations. AIAA Paper 81-1020 Proceedings of the AIAA CFD Conference, Palo Alto, Cal., June 22-23, 1981.

[58] Kessler R.: Oszillatorische Konvektion, Dissertation, Universität Karlsruhe 1983.

[59] Kessler R.: Vectorization of the Galerkin method. In [40].

[60] Kightley J.R.: The Conjugate Gradient Method Applied to Turbulent Flow Calculations. In: Proc. 6 GAMM-Conf. Numer. Methods in Fluid Mechanics (Eds: Rues D. and Kordulle W.) Vieweg-Publ. 1986.

[61] Kordulla W., MacCormack R.W.: Transonic-flow computations using an explicit-implicit method. Proc 8th Int. Conf. Num. Methods in Fluid Dynamics, Lecture Notes in Physics 170, Springer, Berlin, 1982 420-426.

[62] Kordulla W.: On the Numerical Integration of Euler and Navier-Stokes Equations for Compressible Flow-Some Basic Considerations. DFVLR-IB 221-84 A 13, 1984.

[63] Kordulla W.: MacCormack's methods and vectorization. In [40].

[64] Kordulla W., MacCormack R.W.: A New Predictor-Corrector Schema for the Simulation of Three-Dimensional Compressible Flows with Separation. AIAA-Paper 85-1502, 1985.

[65] Kordulla W.: On the Efficient Use of Large Data Bases in the Numerical Solution of the Navier-Stokes Equations on a CRAY computer. In: Notes on Numerical Fluid Mechanics 12. Vieweg Verlag 1986.

[66] Kordulla W., Vollmers H., Dallmann U.: Simulation of Three-Dimensional Transonic Flow With Separation Past a Hemisphere-Cylinder Configuration. AGARD CP-412, Paper 31, 1986.

[67] Kordulla W.: Experiences with an Unfactored Implicit Predictor-Corrector Method. Notes on Numerical Fluid Mechanics, Vol. 13, Vieweg Verlag, pp. 185-192, 1986.

[68] Kordulla W.: Using an Unfactored Implicit Predictor-Corrector Method. AIAA Paper 87-423, 1987.

[69] Kordulla W.: International Workshop on Numerical Simulation. Sept. 30 - Oct. 2, 1987, Göttingen. Proceedings to be published in Notes on Numerical Fluid Mechanics, Vieweg.

[70] Kordulla W.: Integration of the Navier-Stokes Equation in Finite-Volume Formulation. Von Karman Inst. LS 1987-04.

[71] Kutler P.: A Perspective of Theoretical and Applied CFD. AIAA Paper 83-0037, 1983.

[72] Kuwak D., Chang J.L.C., Shanks S.P., Chakravarthy S.R.: A 3D Incompressible Navier-Stokes Flow Solver Using Primitive Variables, AIAA-Journal 3, 1986, 390-396.

[73] Large-Scale Computing in Aeronautics. AGARD Adv. Report No. 209, 1984.

[74] Li C.P.: Chemistry-Split Techniques for Viscous Reactive Blunt Body Flow Computations. AIAA Paper 87-0282, 1987.

[75] Lomax H.: Some prospects for the future of computational fluid dynamics. AIAA-Paper 81-0994, 1981

[76] Löhner R., Morgan K.: Unstructured Multigrid Methods: First Experiences, INME Report Swansea 1985.

[77] MacCormack R.W.: The effect of viscosity in hypervelocity impact cratering. AIAA Paper 69-354 1969.

[78] MacCormack R.W., Stevens K.G.: Fluid dynamics applications of the ILLIAC IV computer. In: Computational Methods and Problems in aeronautical fluid dynamics (Hewitt, ed.), Acad. Press, New York 1976, 448-465.

[79] MacCormack R.W.: A numerical method for solving the equations of compressible viscous flow. AIAA-Paper 81-110 1981, see also AIAA J. 20 1982, 1275-1281.

[80] MacCormack R.W.: Current Status of Numerical Solutions of the Navier-Stokes Equations. AIAA Paper 85-0032, 1985.

[81] Matsuno K.: A vector-oriented finite difference scheme for calculating 3-D compressible laminar and turbulent boundary layers on practical wing configurations. AIAA- Paper 81-1020 1981.

[82] Meiburg E.: Vectorization of the Direct Monte-Carlo Simulation. In [40].

[83] Miyakawa J., Takanashi S., Fujii K., Amano K.: Searching the Horizon of Navier-Stokes Simulation of Transonic Aircraft. AIAA Paper 87-0524, 1987.

[84] Mueller B.: Vectorization of the Implicit Beam and Warming Scheme. In [40].

[85] Mueller B.: Calculation of axisymmetric laminar supersonic flow over blunt bodies, DFVLR Report 1984.

[86] Mueller B., Rizzi A.: Runge-Kutta Finite-Volume Simulation of Laminar Transonic Flow Over a NACA 0012 Airfoil Using the Navier-Stokes Equations. EFA TN 1986-60, 1987.

[87] Mueller-Wichards D., Gentzsch W.: Performance comparisons among several parallel and vector computers on a set of fluid flow problems. DFVLR IB 262-82 R 01 Report, 1982.

[88] Nakahashi K., Obayashi S.: FDM-FEM Zonal Approach for Viscous Flow Computations Over Multiple-Bodies. AIAA Paper 87-0606, 1987.

[89] Napolitano M., Walters R.W.: An Incremental Block-Line-Gauß-Seidel Method for the Navier-Stokes Equations. AIAA Paper 85-0033, 1985.

[90] Newsome R.W., Kandil O.A.: Vortical Flow Aerodynamics-Physical Aspects and Numerical Simulation. AIAA 87-0205. 1987.

[91] Obayashi S., Fujii K., Takanashi S.: Toward the Navier-Stokes Analysis of Transport Aircraft Configurations. AIAA 87-0428, 1987.

[92] Pan D., Pulliam T.H.: The Computation of Steady 3D Separated Flows Over Aerodynamic Bodies at Incidence and Yaw. AIAA-Paper 86-0109, 1986.

[93] Pulliam T.H., Steger J.L.: On implicit finite-difference simulations of 3D flow, AIAA-Paper 78-10 1978.

[94] Pulliam T.H., Lomax H.: Simulation of 3-D compressible viscous flow on the ILLIAC IV computer. AIAA-Paper 79-0206 1979.

[95] Pulliam T.H., Steger J.L.: Recent Improvements in Efficiency, Accuracy, and Convergence for Implicit AF Algorithms. AIAA-Paper 85-0360, 1985.

[96] Pulliam T.H., Jesperson D.C., Barth T.J.: Navier-Stokes Computations for Circulation Controlled Airfoils. AIAA- Paper 85-1587, 1985.
C

[97] Pulliam T.H.: Efficient Solution Methods for the Navier- Stokes Equations. Von Karman Inst. Brussels 1986.

[98] Rai M.M.: Navier-Stokes Simulations of Blade-Vortex Interaction Using High-Order Accurate Upwind Scheme. AIAA Paper 87-0543, 1987.

[99] Redhed D.D., Chen A.W.: New approach to the 3-D transonic flow analysis using the STAR-100 computer. AIAA-Journal 17 1979, 98-99.

[100] Reznick S.G., Flores J.: Strake-Generated Vortex Interactions for a Fighter-Like Configuration. AIAA-Paper 87-0589, 1987.

[101] Rizzi A.: Vector coding the finite volume procedure for the CYBER 205. VKI-Lecture Series 1983-04.

[102] Rogers S.E., Kwak D., Chang J.L.C.: Numerical Solution of the Incompressible Navier-Stokes Equations in 3D Curvilinear Coordinates. NASA TM 86840, 1986.

[103] Rogers S.E., Kwak D., Kaul U.K.: A Numerical Study of 3D Incompressible Flow Around Multiple Posts. AIAA-Paper 86- 0353, 1986.

[104] Rogers S.E., Chang J.L.C., Park C.A., Kwak D.: A Diagonal Algorithm for the Method of Pseudocompressibility. AIAA- Paper 86-1060, 1986.

[105] Rubbert P.E.: The Impact of Computational Methods on Aircraft Design. CRAY-Channels 6, 1984 No.4, 2-5.

[106] Schmatz M.A.: Simulation of Viscous Flows by Zonal Solutions of Euler, Boundary-Layer and Navier-Stokes Equations. Paper presented at the DGLR-Jahrestagung Okt. 1986, Munich.

[107] Schoenauer W., Gentzsch W. (Eds.): The Efficient Use of Vector Computers with Emphasis on CFD. Vieweg Publ., F.R.G. 1986.

[108] Schröder W., Hänel D.: An Unfactored Implicit Scheme with Multigrid Acceleration for the Solution of the Navier-Stokes Equations. To be published in Computers and Fluids, 1987.

[109] Schwamborn D.: Vectorization of an Implicit Finite Difference Method for the Solution of the Boundary Layer Equation. In [40].

[110] Schwamborn D., Reister H.: Entwicklung einer Navier-Stokes Lösung für den DFVLR-F5 Flügel. DFVLR-IB 221-86 A 08. 1986.

[111] Scott J.N.: Numerical Simulation of Time-Dependent Viscous Flows in Aerospace Propulsion Systems. CRAY Channels 8, 1986 No.2, 6-11.

[112] Shang J.S., Buning P.G., Hankey W.L., Wirth M.C., Calahan D.A., Ames W.: Numerical solution of the 3-D Navier-Stokes equations on the CRAY-1 computer. Proc. Scientific Comp. Inf. Exchange Meeting 1979, 159-166.

[113] Shang J.S., Buning P.G., Hankey W.L., Wirth M.C.: Performance of a vectorized three-dimensional Navier- Stokes code on the CRAY-1 computer. AIAA-Journal 18 1980, 1073-1078.

[114] Shang J.S.: Numerical simulation of wing-fuselage interference. AIAA-Paper 81-0048 1981.

[115] Smith R.E., Pitts J.I., Lambiotte J.J.: A vectorization of the Jameson-Caughey transonic swept-wing computer program FLO-22 for the STAR-100 computer. NASA Techn. Memorandum TM-78665 1978.

[116] South J.C., Keller J.D., Hafez M.M.: Vector processor algorithms for transonic flow calculations. AIAA-Journal 18 1980, 786-792.

[117] Spradley L.W., Stalnaker J.F., Ratliff A.W.; Hyperbolic/parabolic development for the GIM-STAR code. NASA Contractor Report 3369 1980.

[118] Spradley L.W., Stalnaker J.F., Ratliff A.W.: Solution of the three-dimensional Navier-Stokes equations on a vector processor. AIAA-Journal 19 1981, 1302-1308.

[119] Swisshelm J.M., Johnson G.M.: Numerical Simulation of 3D Flowfields Using the CYBER 205. ICS Techn. Report 85002, 1985.

[120] Swisshelm J.M., Johnson G.M.: Parallel Computation of Euler and Navier-Stokes Flows. ICS Techn. Report 85003, 1985.

[121] Thomas J.L., Walters R.W.: Upwind Relaxation Algorithms for the Navier-Stokes Equations. AIAA Paper 85-1501-CP 1985.

[122] Thomas J.L., Taylor S.L., Anderson W.K.: Navier-Stokes Computations of Vortical Flows Over Low Aspect Ratio Wings. AIAA Paper 87-0207, 1987.

[123] Thompson J.F.: A Survey of Grid Generation Techniques in CFD. AIAA-Paper 83-0447, 1983.

[124] Thompson J.F.: A Composite Grid Generation Code for General 3-D Regions. AIAA Paper 87-0275, 1987.

[125] Vadyak J., Smith M.J., Schuster D.M, Weed R.: Simulation of External Flow fields Using a 3-D Euler/Navier-Stokes Algorithm. AIAA Paper 87-0484, 1987.

[126] Vuong S.T., Coakley T.J.: Modeling of Turbulence for Hypersonic Flows With and Without Separation. AIAA Paper 87-0286, 1987.

[127] Wu C-T., Ferziger J.H., Chapman D.R.: Simulation and Modeling of Homogeneous Compressed Turbulence. NASA Techn. Report TF-21, 1985.

[128] Ying S.X., Steger J.L., Schiff L.B., Baganoff D.: Numerical Simulation of Unsteady, Viscous, High Angle of Attack Flows Using a Partially Flux-Split Algorithm. AIAA Paper 86-2179, 1986.

[129] Yoon S., Jameson A.: An LU-SSOR Scheme for the Euler and Navier-Stokes Equations. AIAA Paper 87-0600, 1987.

[130] Zabolitzky J.G.: Vector programming of Monte-Carlo and numerical problems. In: Proc. of the 1982 Conf. on CYBER 200 in Bochum (Bernutat-Buchmann U., Ehrlich H., Schlosser K.-H., eds.), Bochumer Schriften zur Parallelen Datenverarbeitung 1982, 165-174.

Additional References (provided by Dr. J. Steger)

[131] MacCormack, R. and Baldwin, B., "A Numerical Method for Solving the Navier-Stokes Equations With Application to Shock-Boundary Layer Interactions," AIAA Paper 75-1, 1975.

[132] MacCormack, R., "A Numerical Method for Solving the Equations of Compressible Viscous Flow," AIAA Paper 81-110, 1981.

[133] Pulliam, T. H., and Steger, J. L., "On Implicit Finite Difference Simulations of Three-Dimensional Flows," AIAA J. Vol. 18, 1979.

[134] Thomas, J. L., Taylor, S. L., and Anderson, W. K., "Navier/Stokes Computations of Vortical Flows Over Low Aspect Ratio Wings," AIAA Paper 87-207, 1987.

[135] Shang, J. S., and Scherr, S. J., "Navier/Stokes Solution of the Flow Field Around a Complete Airplane," AIAA Paper 85-1509, 1985.

[136] Rai, M. M., "Navier/Stokes Simulations of Rotor-Stator Interaction Using Patched and Overlaid Grids," AIAA Paper 85-1519, 1985.

[137] Shankar, V., and Chakravarthy, S., "Development and Application of Unified Algorithms for Problems in Computational Science," NASA CP 2454, 1987.

[138] Chaussee, D., Rizk, Y., and Buning, P., "Viscous Computation of a Space Shuttle Flow Field," Ninth International Conference on Numerical Methods in Fluid Dynamics, June 1984. (See also NASA TM 85977, 1984.)

[139] Flores, J., Holst, T., Kaynak, U., Grundy, K., and Thomas, S., "Transonic Navier/Stokes Wing Solution Using a Zonal Approach. Part 1. Solution Methodology and Code Validation," AGARD CP 412, 1986.

[140] MacCormack, R., "Current Status of Numerical Solutions of the Navier/Stokes Equations," AIAA Paper 85-032, 1985.

[141] Benek, J. A., Donegan, T. L., and Suhs, N. E., "Extended Chimera Grid Embedding Scheme with Applications to Viscous Flows," Proceedings of the AIAA CFD Conference, 1987.

[142] Newsome, R. W., and Adams, M. S., "Numerical Simulation of Vortical Flow over an Elliptical Body Missile at High Angle of Attack," AIAA Paper 86-559, 1986.

[143] Fujii, K., and Schiff, L., "Numerical Simulation over a Strake-Delta Wing," AIAA Paper 87-1229, 1987.

[144] Coakley, T., "Numerical Method for Gasdynamics Combining Characteristics and Conservation Concepts," AIAA Paper 81-1257, 1981.

[145] Viviand, H., "Conservation Forms of Gas Dynamic Equations," La Recherche Aerospatiale, No. 1, 1974.

[146] Jameson, A., "Solutions of the Euler Equations for Two Dimensional Transonic Flow by a Multigrid Method," Applied Math. and Computations, Vol. 13, 1983.

[147] Jespersen, D. C., "Recent Developments in Multigrid Methods for the Steady Euler Equations," VKI Lecture Series on CFD, Belgium, 1984.

[148] Van Dalsem, W., and Steger, J., "The Fortified Navier/Stokes Approach," Workshop on CFD, University of California-Davis, June 1986.

[149] Beam, R. M., and Warming, R. F., "An Implicit Finite Difference Algorithm for Hyperbolic Systems in Conservation Law Form," J. Comp Phy, Vol. 22, 1976.

[150] Warming, R., and Beam, R., "On the Construction and Application of Implicit Factored Schemes for Conservation Laws," SIAM-AMS Proceedings, Vol. 11, 1977.

[151] Pulliam, T., and Steger, J., "Recent Improvements in Efficiency, Accuracy, and Convergence of an Implicit Approximate Factorization Algorithm," AIAA Paper 85-360, 1985.

[152] Steger, J. and Warming, R., "Flux Vector Splitting of the Inviscid Gasdynamic Equations with Applications to Finite Difference Methods," J. Comp Phys, Vol. 40, 1981.

[153] Van Leer, B., "Flux Vector Splitting for the Euler Equations," Proc. 8th International Conference on Numerical Methods in Fluid Dynamics, Springer, 1982.

[154] Anderson, W. K., Thomas, J. L., and Van Leer, B., "A Comparison of Finite Volume Flux Vector Splittings for the Euler Equations," AIAA Paper 85-122, 1985.

[155] Thomas, J., and Walters, R., "Upwind Relaxation Algorithms for the Navier/Stokes Equations," AIAA Paper 85-1501, 1985.

[156] Ying, S., Steger, J., Schiff, L., and Baganoff, D., "Numerical Simulation of Unsteady Viscous High Angle of Attack Flows Using a Partially Flux-Split Algorithm," AIAA Paper 86-2179, 1986.

[157] Chakravarthy, S., "Euler Equations-Implicit Schemes and Implicit Boundary Conditions," AIAA Paper 82-22;, 1982.

[158] Chaussee, D., and Pulliam, T., "A Diagonal Form of an Implicit Approximate Factorization Algorithm with Application to a Two Dimensional Inlet," AIAA J., Vol. 19, 1981.

[159] Barth, T., and Steger, J., "An Efficient Approximate Factorization Implicit Scheme for the Equations of Gasdynamics," NASA TM 85957, 1984.

[160] Gentzsch, W., "Vectorization of Computer Programs with Application to Computational Fluid Dynamics," Vieweg, 1984.

[161] Reddy, K., "Pseudospectral Approximation in a Three Dimensional Navier/Stokes Code," AIAA J., Vol. 21, 1983.

[162] Yee, H., Warming, R., and Harten, A., "Implicit Total Variation Diminishing (TVD) Schemes for Steady-State Calculations," NASA TM 84342, 1983.

[163] Yee, H., and Harten, A., "Implicit TVD Schemes for Hyperbolic Conservation Laws in Curvilinear Coordinates," AIAA Paper 85-1513, 1985.

[164] Chow, L., Pulliam, T., and Steger, J., "A General Perturbation Approach for Computational Fluid Dynamics," AIAA J., Vol. 22, 1984.

[165] Israel, M., and Ungarish, M., "Improvements of Numerical Solutions by Incorporation of Approximate Solutions Applied to Rotating Compressible Flows," Proceedings 7th International Conference on Numerical Methods in Fluid Dynamics, Springer, 1980.

[166] Nakahashi, K., and Obayashi, S., "Viscous Flow Computations Using a Composite Grid," Proceedings AIAA 8th CFD Conference, 1987.

APPENDIX A
AN INFORMAL GLOSSARY OF TERMS USED IN SUPERCOMPUTING

A.1 INTRODUCTION

This informal glossary of terms associated with supercomputing evolved casually. The primary draft was made by K. Neves in 1986 for the IEEE Subcommittee on Supercomputing. This draft benefited greatly from review and revision by fellow members of the committee (namely, A. Brenner, D. Lawrie, S. Perrenod, J. Riganati, S. Saphier, and P. Schneck.) In addition, terms have been added and revised by the authors of this monograph. Any rapidly advancing field generates its own new terminology and jargon. The new terms arise by applying new meanings to older terms as well as through the "coining" of new terminology. The process is dynamic and often results in dual meanings of commonly used words and sometimes just plain fuzzy definitions. Nevertheless, the authors believe this tabulation will a) help both novices and experts to better communicate, and b) provide input to more formal dictionary/definition writing processes.

A.2 TERMS AND DEFINITIONS

AUTOVECTORIZATION: The ability of a compiler to generate object code which utilizes vector hardware, vector firmware, or vector microcode, starting from standard (e.g. Fortran 77) source language.

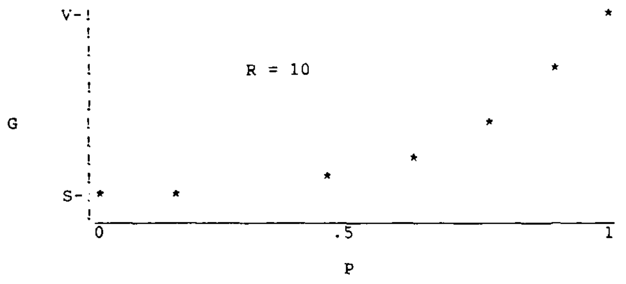
AMDAHL'S LAW: When dealing with high performance hardware, one often encounters concurrency in computation. This is usually achieved through pipelined arithmetic units or parallel cpus. In either case, the achievable performance is a function of the amount of the computation that can be placed in the high performance mode. So, for example, if the potential speed up is 10 to one, running in parallel or in vector mode, then if all the computation is moved to the high performance process, things will run in one tenth the time. However, if only 75% were moved to the high performance process, the total time would be reduced to about 32.5% of its former value. This domination of the slower process is often referred to as Amdahl's Law. A more formal definition for parallel and vector computers is given by the following formulae:

Amdahl's Law (Vector Computers):

Let V be the vector speed of a process and S be the scalar speed. Then the final gain in speed, G, of a process that is P percent vectorized is given by

$$G = [(1-P) + P/R]^{-1}, \quad \text{where } R = V/S$$

A plot of G versus P when V is ten times S reveals the familiar Amdahl-Curve.

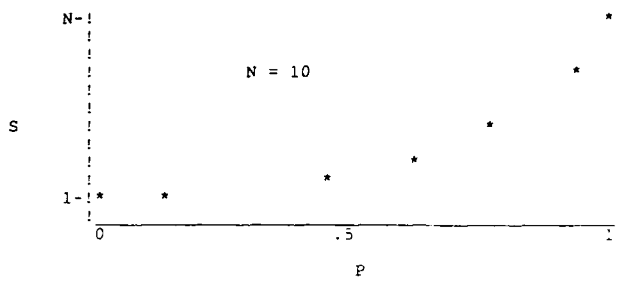


Amdahl's Law (Parallel Computers):

Let N be the number of parallel processors, and P the percentage of the work performed simultaneously (ignoring any new overhead introduced) on the N processors. Then the speedup, S attained in performance is given by

$$S = \frac{\text{partial parallel speed}}{\text{single processor speed}} = (1 - P + P/N)^{-1}$$

If one fixes N and plots S versus P, the percentage of parallelization, the familiar curve below is obtained.



ARRAY PROCESSORS: These are attached processors. That is, they usually require a host computer to interface to the user. They are essentially "subroutine boxes". Often they are programmable through microcode, and sometimes higher languages. They achieve high performance relative to their hosts on specific tasks, e.g., seismic processing. Their computing engines are typically composed of pipelined arithmetic units and over-lapped functional units.

BANK CONFLICT: Since memory chip speeds are relatively slow when required to deliver a single word, supercomputer memories are placed in a large number (usually a power of 2) of independent banks. A vector laid out contiguously in memory (one component per successive bank) can be accessed at one word per cycle despite the intrinsic slowness of memory chips to deliver a single word. The result is the "pipelined" delivery of vectors component words at high bandwidth (after first word initial latency). The "bank cycle time" is the number of clock periods (or seconds) a given bank must wait before a successive access can be fulfilled. When the number of banks is a power of 2, then vectors requiring strides of a power of 2 can often run into bank busy-wait situation. This often termed a bank conflict. As an example, imagine a 64-bank memory with an 8-clock cycle time. If one accesses every 16th element in a contiguously stored vector, a bank conflict will occur in for cycles (clock periods). The result would be that every four fetches, a four cycle wait would occur causing an overall reduction in performance of a factor of two. Such conflicts can also occur randomly and more frequently in multiple CPU machines that share a common memory. It is also possible for one CPU to access banks so efficiently as to block out the other CPUs.

CACHE: Large memory requirements often lead to dense but slow memories. Memory throughput is high for large amounts of data, but for individual or small amounts of data, the fetch times can be very long. To overcome this computer architects use smaller interface memories with better "fetch" speeds. These are often called CACHE memories. The term is more often used when these memories are a required interface to, say, main memory. If the required data is already stored in the cache, fetches are fast. If the required data is not in the cache, a "cache miss" occurs and results in cache being refilled from main memory at the expense of time. Their use is usually made transparent to the user and takes advantage of the fact that a reference to a given area of main memory for one piece of data or instruction is usually closely followed by several additional references to that same area for other data or instructions; thus, maximizing the potential for cache "hits." This "prefetch" process is managed by the firmware of the computer system.

CHAINING: The ability to take the results of a vector operation and use them directly as input operands in a further vector instruction; thus alleviating the need for additional store and fetch instructions. This chaining or "linking" of two vector floating point operations, for example, could double the asymptotic MFLOP rate.

COOLING TECHNOLOGY: At the high performance end of computing, speed is related directly to the density of chip packaging. Dense packaging usually produces unusual cooling requirements. The technology of cooling supercomputers takes many forms from air cooling (Fujitsu VP 200, Hitachi S-820), water cooling (NEC SX/2), freon (CRAY 1, X-MP), flourinert submersion (CRAY-2), and cryogenic cooling via liquid nitrogen immersion (ETA-10). Cooling technology has become a critical part of advanced computer design.

COMPRESS/INDEX: This is a "vector" operation which is used to deal with the non-zeros of a large vector with relatively few non-zeros. The location of the non-zeros is indicated by an index vector (usually a bit vector of the same length, in bits, as the full vector, in words). The COMPRESS operations gathers the non-zeros into a dense vector according to the index vector. This operation is usually order M, where M is the length of the full vector, and the resulting dense vector is of length N, the number of non-zeros. This is different than the true gather/scatter operations which are order N (see gather/scatter).

DRYSTONES: Established as a non-numeric counterpart to the Whetstone benchmark suite. (See Whetstone.)

DISK STRIPING: Multiplexing or interleaving a data set (disk file) across 2 or more disk drives to enhance I/O performance. The performance gain is at most equal to the minimum of the number of drives and channels used.

DISTRIBUTED PROCESSING: Processing on a number of computers networked in some fashion. One of the distinguishing features of distributed processing is the absence of a shared main memory. One typically infers the computers are of different relative power and function. For example, a supercomputer, a mini-computer, and a work station all providing computation for a single application in such a way that the process is distributed over all the vehicles as appropriate.

FIFTH GENERATION: The Japanese AI initiative to build a super AI processor with high LIP throughput. Also the next technological "generation".

FLOPS: Floating point (arithmetic) operations per second.

GATHER/SCATTER: The operations related to large sparse data structures. A full vector with relatively few non-zeros is transformed into a vector with only the non-zeros with a "gather" operation. The full vector or one with the same structure is built from the inverse operation, the "scatter". The process is accomplished with an index vector. The index vector is usually of length N, the number of non-zeros with each component being the relative location in the full vector. This is in sharp contrast to the index vector of the "compress" operation where the index vector is bit oriented of length M, the length of the full vector, with only ones or zeros in each component to indicate whether there is an associated non-zero in the full vector.

GLOBAL MEMORY: Main memory accessible by all processors or CPUs.

GRANULARITY: Most applications can be broken down into sub-processes. Fine granularity is illustrated by execution of statement or small loop iterations as separate processes, where coarse granularity would involve subroutines or sets of subroutines as a group as separate process. The term is often used in parallel processing to indicate independent processes which could be distributed to multiple CPUs. Typically the more processes there are, the "finer" the granularity and the more overhead in keeping track of them. Granularity can also be considered to be related to the temporal duration of a "chunk" of work. It is not only the number of processes but how much work each process does relative to the time of synchronization that determines the overhead and reduces speedup figures. Since the granularity is related to the overhead of synchronization, it is necessarily machine dependent. Granularity could be defined as the ratio of task computation time versus synchronization time. If the ratio is large, the granularity is high and the expense of synchronization is minimized.

HYPERCUBE ARCHITECTURE: multiple CPU architecture with 2**N processors. Each CPU has N nearest neighbors in a manner similar to a "hypercube" where each corner has N edges. The 2**3 machine would have 8 CPUs arranged at the corners of a cube connected by the edges, as illustrated below.

```
                110                        111
                  *-------------------------*
                 /!                        /!
                / !                       / !
               /  !                      /  !
              /                         /   !
             /    !                    /    !
            /     !                   /     !
     010  *--------------------------*  011 !
          !                          !      !
          !  100 *  -  -  -  -  -  - ! -  - * 101
          !                          !     /
          !     /                    !    /
          !                          !   /
          !   /                      !  /
          !                          ! /
          ! /                        !/
          !                          !
          *--------------------------*
     000                                001
                 Hypercube, N=3
```

INTERACTIVE VECTORIZER: An interactive program to aid a user in vectorizing his source code. The program usually analyzes the source for loops and sequences of operations that can be accomplished using vector instructions or macros. When obvious obstructions to vectorization are found the user is informed through interaction of the problem. Quite often the user can indicate that a vector reference which has a potential recursive reference is "safe" or the user can remove an "IF-test", branch or subroutine call, to achieve vectorization.

INTERNAL THROUGHPUT RATE, ITR: actual job stream processing time including all aspects of the computer, I/O, O/S etc.

INTER-PROCESSOR CONTENTION: Contention by multiple CPUs for shared system resources. For example, in global memory architectures memory bank conflicts for a user's code are caused by other processors running completely independent applications.

LIPS: Logical inferences per second. One LIPS is essentially a PROLOG procedure call and was originally defined by the Japanese when they announced their Fifth Generation program in the Fall of 1981.

LOCAL MEMORY: The memory associated with a single CPU in a multiple CPU architecture, or memory associated with a local node in a distributed system.

LOOP UNROLLING: An optimization technique valid for both scalar and vector architectures. The iterations of an inner-loop are decreased by a factor of 2 or more by by explicit inclusion of the very next or next several iterations. This allows the compiler to make better use of the registers (avoiding some memory references) and better overlap operations. On vector machines loop unrolling may either improve or degrade performance. This involves a tradeoff between overlap and register use on the one hand and vector length on the other.

MFLOPS, MEGA FLOPS: Millions of floating-point (arithmetic) operations per second. A common rating of supercomputers (and vector instruction machines). Some have used the euphemism "macho FLOP" to indicate the fact that MFLOP ratings are performance measures that indicate what a machine cannot exceed in performance, rather than indicate actual performance. There is generally a large variance between a supercomputer's peak MFLOP rating and its typical sustained MFLOP performance on an actual algorithm or application. It is also worthy to note that different computers behave differently when it comes to achieving "close" to peak performance. On one computer it may be very difficult to design an algorithm to achieve 30% of peak, while on another the same algorithm can be implemented easily and achieve 70% peak performance.

MAIN MEMORY: A Level of random access memory which lies between cache/register memory on the one end and extended random access memory on the other. Main memory has higher capacity, but is slower than cache or registers, and is less capacity, but faster access than extended random access memory. Unfortunately, some newer supercomputer models have introduced another layer or tier of memory structure and what is cache, main, extended is getting somewhat muddled.

MINISUPERCOMPUTER: a machine with roughly 1/10 to 1/2 the performance of a supercomputer at roughly 1/10 the price. Minisupers use a blend of minicomputer technology and supercomputer architectural "tricks" (i.e. pipelining, vector instructions, parallel CPUs) to achieve attractive price performance characteristics.

MIMD: Multiple instruction stream/multiple data stream architecture (or process). In an MIMD machine multiple instruction streams are simultaneously in execution. Each single instruction may handle multiple data elements (e.g., one or more vectors in a vector machine). While single processor vector computers are able to operate in MIMD mode due to overlapped functional units, this terminology is more generally used to refer to multiprocessor machines.

MIPS: Millions of instructions per second.

MOPS: Millions of operations per second.

MULTIPROCESSING: See Multitasking.

MULTIPROGRAMMING: See Multitasking.

MULTIPROCESSOR: A single computer system with more than one CPU. Usually the CPUs would be more tightly coupled than simply sharing a local area network. For example, CPUs sharing main memory or a file system would be called a multiprocessor.

MULTITASKING: The terms multiprogramming, multiprocessing, and multitasking are often used interchangeably (with a notable lack of precision) to describe three different concepts: The use of more than one processor; the commingling of different programs on one or more processors; and the execution of multiple tasks from the same program on one or more processors. No rigorous definitions exist, but generally, the term "multiprogramming" refers to the ability of a computer to commingle more than one program on at least one CPU. "Multitasking" is often used to mean the same thing, but more recently has also been used to mean the simultaneous execution of several tasks from the same program on two or more CPUs. "Multiprocessing" is also used to indicate the ability of a computer to commingle jobs on one or more CPUs. More recently, it is also used as a synonym for multitasking.

$N_{1/2}$ (N SUB A HALF): The length of a vector required (for a given vector operation or sequence of operations) to achieve one half the peak performance rate. (Defined by Roger Hockney along with a number of other interesting parameters in a monograph on Parallel Processing.) Large N SUB A HALVES indicated a great deal of overhead associated with vector startup. A rule of thumb is that if the average vector length is 3 times N SUB A HALF, one is being very efficient; while if the vector length is less than N SUB A HALF, one is being inefficient. (All quite qualitative)

OPTIMIZE: Achieve peak possible performance for a process, or a number of related processes. More loosely applied whenever a procedure improves, even modestly, a previous attempt.

PARALLEL PROCESSING: *Processing with more than one CPU on a single application simultan-*eously.

PARALLELIZATION: The process of achieve a high percentage of the CPU time expended in parallel. That is, minimizing idle CPU time in a parallel processing environment. For a specific program, parallelization refers to the splitting of its execution-tasks among available CPUs.

PARTITIONING: Restructuring a program or algorithm in semi-independent computational segments to take advantage of multiple CPUs simultaneously. The goal is to achieve roughly equivalent work in each segment with minimal need for inter-segment communication. It is also worthwhile to have fewer segments than CPUs on a dedicated system.

PERCENTAGE PARALLELIZATION: The percent of CPU expenditure being processed in parallel on a single job. It is usually impossible to achieve 100% of an application's processing time to be equally shared on all CPUs. A related question, is a reliable measure of efficiency which is beyond the scope of this discussion.

PERCENTAGE VECTORIZATION: Percentage of an application running in "vector mode". This may be calculated in two ways: 1) as a percentage of CPU time; 2) as that percentage of lines of code (usually Fortran) which are coded into vector instructions. The two usages are not consistent and may give very different results. The first definition will lead to clear performance improvement as measured by CPU time, while the second method is confined to measure the "success rate" of the compiler in converting scalar code to vector code (however dubious). The former is a hardware performance measure, and the latter is a compiler performance measure.

PIPELINING: Any execution of a sequence of data sets (possibly instructions) by a single processor, in such a way that subsequent data elements (instructions) in the sequence can begin execution before previous elements have completed execution, with all such elements executing at the same time--an assembly line approach. In modern supercomputers the floating point operations are often pipelined along with memory fetches and stores of the "vector" data sets. (The Denelcor HEP is an example of pipelined instruction sets, the Japanese Supercomputers, the 3090/VF, CRAY and Cyber 200 series give examples of pipelined arithmetic units.)

PRIMARY MEMORY: Main memory accessible by the CPU(s) without using I O processes.

R-INFINITY: The asymptotic rate of a vector operation as vector length approaches infinity (originally defined by Hockney and Jessophe in their text on parallel processing.)

RECURRENCE: A dependency in a DO-loop whereby a result depends on completion of the previous iteration of the loop. Such references inhibit vectorization. For example,

$$A(I) = A(I-1) + B(I)$$

in a loop on I, would not be vectorizable on most vector computers (without marked degradation in performance). This is ot a axiom or law, simply a fact. There are manufacturers toying with the idea of vectorizing simple recurrences of this type. In fact, several Japanese compiler address the construct with a combination of hardware and software. Quite often, however, such recurrences can be avoided or done simultaneously to achieve vectorizable computations.

RISC: RISC refers to a philosophy of instruction set design where a small number of simple, fast instructions are implemented instead of a larger number of slower, more complex instructions. RISC is becoming a marketing term, and considerable controversy exists as to which machines adhere to the RISC philosophy and which do not.

SECONDARY MEMORY: Often larger and slower memory than primary memory (see primary memory) and often requires special instructions (e.g. I/O instructions) to access.

SIMD: Single instruction stream/multiple data stream. Characterizes most of today's vector computers. A single instruction initiates a process that causes streams of data and results to be set in motion. The term is also applicable to parallel processors where one instruction causes N processors to (often synchronously) perform the same operation on perhaps different pieces of data (e.g., ILLIAC).

SISD: Single instruction stream/single data stream. Traditional scalar architecture.

SPEED UP: Term often used related to vector and/or parallel processing. The idea is to give a factor of performance improvement over pure scalar performance. The reported numbers are often misleading due to inconsistency in reporting the speedup over a revised process running in scalar mode or the original process running in scalar mode. The term is usually applied to performance on the same CPU versus multiple but identical CPUs or vector vs. scalar process on the same machine.

STRIDE: A term often used relative to vector storage. A mathematical "vector" is simply an array of numbers. For a computer, an array of numbers is simply data whose location is prescribed according to some formula. Vector computers often reference "computer" vectors according to differing conventions. Some computers, notably a CYBER 205, refer to vectors by first word location and length. This is "contiguous" storage of a vector. Unfortunately, many applications in matrix analysis require the fetch and store of vectors whose components do not reside contiguously in memory. An example is the row of a column-stored matrix. The matrix is stored contiguously, but the rows have elements that are spaced in memory by a STRIDE of N, the dimension of the matrix. This is often termed "regularly stored with stride of N." To complete the picture, some vector computers allow vector fetch and stores to occur with randomly stored vectors, i.e., first word and an index vector which maps the relative location of successive components. This is often useful in storing the non-zero elements of a sparse vector. The term is mnemonic, being derived from the concept of walking (striding) through the data from one non-contiguous location to the next.

SUPERCOMPUTER(S): The class of general purpose computers that are both faster than their commercial competitors AND have sufficient central memory to store the problem sets for which they are designed. The issue of "computer power" in large-scale scientific processing is a complex topic. Computer memory, throughput, computational rates, and a host of related computer attributes contribute to performance. Consequently, a quantitative measure of computer power does not exist, and a precise definition of supercomputers is difficult.

TRUE RATIO: A frequent bottleneck to vectorization of scientific programs is the Fortran IF-test. Quite often the successful branch is executed only once in awhile in what could be a highly vectorizable "loop". The term TRUE RATIO is used to characterize the frequency the branched condition occurs. If the true-ratio is small, some compilers can take effective action. The problem is that the TRUE RATIO is often data dependent and can't effectively be dealt with automatically.

VECTOR: An array of numbers whose location is prescribed according to some formula (e.g., contiguously or randomly). see STRIDE. One may distinguish a computer vector from a mathematical vector--the latter is simply a set of numbers (components) with no conditions on their retrievability from computer memory.

VECTOR PROCESSING: Modern supercomputers achieve speed through pipelined arithmetic units. This coupled with instructions designed to process vectors or arrays of numbers rather than each data pair one at a time, leads to great performance improvements. Since computers with this design deal with vectors, the "art" of using such machines has been called VECTOR PROCESSING.

VECTORIZATION: The act of tuning an application code to take advantage of vector architecture; see percentage of vectorization.

WHETSTONE: A benchmark suite established and maintained by CCTA (an element of the British Post Office). Based on experience with the KDF-9 (and its Whetstone system), this benchmark set was established. It exists in two forms -- Whetstone I for 32-bit floating point operations and Whetstone II for 64-bit floating point operations. The code is highly floating point intensive, yet not very amenable to vectorization. Large machines such as the CRAY series can execute some millions of Whetstones per second.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-AG-311	ISBN 92-835-0448-8	UNCLASSIFIED

5. Originator Advisory Group for Aerospace Research and Development
North Atlantic Treaty Organization
7 rue Ancelle, 92200 Neuilly sur Seine, France

6. Title
COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS

7. Presented at

8. Author(s)/Editor(s)	9. Date
W.Gentzsch and K.W.Neves Edited by H.Yoshihara	March 1988

10. Author's/Editor's Address	11. Pages
Various	196

12. Distribution Statement This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.

13. Keywords/Descriptors

Navier/Stokes	Vector programming
Algorithms	Benchmarking
Supercomputers	Mini-supercomputers

14. Abstract

Cost-effective vectorization of fluid dynamic codes, in particular the Navier/Stokes Code, is covered relative to the supercomputer architecture. Subjects include current supercomputer architecture; minisupercomputers; impact of hardware on computing; software migration issues; benchmarking; guidelines on Fortran vectorization at the do-loop level; restructuring of basic linear algebra algorithms; and restructuring guidelines for basic fluid dynamic codes. A glossary of supercomputing terms is given in the Appendix.

AGARDograph No.311	AGARD-AG-311
Advisory Group for Aerospace Research and Development, NATO COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS by W.Gentzsch and K.W.Neves, Edited by H.Yoshihara Published March 1988 196 pages Ccst-effective vectorization of fluid dynamic codes, in particular the Na ier/Stokes Code, is covered relative to the supercomputer architecture. Subjects include current supercomputer architecture; minisupercomputers; impact of hardware on computing; software migration issues; benchmarking; guidelines on Fortran vectorization at the do-loop level; restructuring of basic linear algebra P.T.O	Navier/Stokes Algorithms Supercomputers Vector programming Benchmarking Mini-supercomputing

AGARDograph No.311	AGARD-AG-311
Advisory Group for Aerospace Research and Development, NATO COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS by W.Gentzsch and K.W.Neves, Edited by H.Yoshihara Published March 1988 196 pages Cost-effective vectorization of fluid dynamic codes, in particular the Navier/Stokes Code, is covered relative to the supercomputer architecture. Subjects include current supercomputer architecture; minisupercomputers; impact of hardware on computing; software migration issues; benchmarking; guidelines on Fortran vectorization at the do-loop level; restructuring of basic linear algebra P.T.O	Navier/Stokes Algorithms Supercomputers Vector programming Benchmarking Mini-supercomputing

AGARDograph No.311	AGARD-AG-311
Advisory Group for Aerospace Research and Development, NATO COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS by W.Gentzsch and K.W.Neves, Edited by H.Yoshihara Published March 1988 196 pages Cost-effective vectorization of fluid dynamic codes, in particular the Navier/Stokes Code, is covered relative to the supercomputer architecture. Subjects include current supercomputer architecture; minisupercomputers; impact of hardware on computing; software migration issues; benchmarking; guidelines on Fortran vectorization at the do-loop level; restructuring of basic linear algebra P.T.O	Navier/Stokes Algorithms Supercomputers Vector programming Benchmarking Mini-supercomputing

AGARDograph No.311	AGARD-AG-311
Advisory Group for Aerospace Research and Development, NATO COMPUTATIONAL FLUID DYNAMICS: ALGORITHMS & SUPERCOMPUTERS by W.Gentzsch and K.W.Neves, Edited by H.Yoshihara Published March 1988 196 pages Cost-effective vectorization of fluid dynamic codes, in particular the Navier/Stokes Code, is covered relative to the supercomputer architecture. Subjects include current supercomputer architecture; minisupercomputers; impact of hardware on computing; software migration issues; benchmarking; guidelines on Fortran vectorization at the do-loop level; restructuring of basic linear algebra P.T.O	Navier/Stokes Algorithms Supercomputers Vector programming Benchmarking Mini-supercomputing

algorithms; and restructuring guidelines for basic fluid dynamic codes. A glossary of supercomputing terms is given in the Appendix.

This AGARDograph has been produced at the request of the Fluid Dynamics Panel of AGARD.

ISBN 92-835-0448-8

algorithms; and restructuring guidelines for basic fluid dynamic codes. A glossary of supercomputing terms is given in the Appendix.

This AGARDograph has been produced at the request of the Fluid Dynamics Panel of AGARD.

ISBN 92-835-0448-8

algorithms; and restructuring guidelines for basic fluid dynamic codes. A glossary of supercomputing terms is given in the Appendix.

This AGARDograph has been produced at the request of the Fluid Dynamics Panel of AGARD.

ISBN 92-835-0448-8

algorithms; and restructuring guidelines for basic fluid dynamic codes. A glossary of supercomputing terms is given in the Appendix.

This AGARDograph has been produced at the request of the Fluid Dynamics Panel of AGARD.

ISBN 92-835-0448-8